Pedro Miguel dos Santos Valdevino

Licenciado em Engenharia Informática

Optimização de uma Aplicação Paralela para Simulações de Dinâmica Molecular

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: Professor Doutor Paulo Afonso Lopes, Professor Auxiliar, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa – Departamento de Informática

Júri:

Presidente: Prof. Doutor Luís Manuel Marques da Costa Caires
Arguente: Prof. Doutor Salvador Pinto Abreu
Vogal: Prof. Doutor Paulo Orlando Reis Afonso Lopes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro 2011

Optimização de uma Aplicação Paralela para Simulações de Dinâmica Molecular

Pedro Valdevino

23 de Setembro de 2011

Direito de Cópia

Agradecimentos

Ao Professor Doutor Paulo Afonso Lopes, por ter estado presente sempre que precisei dos seus valiosos ensinamentos. Sem o seu apoio, e sem a sua atenta orientação, esta dissertação não teria sido de todo possível.

À Professora Doutora Leonor Cruzeiro do Departamento de Física da Universidade do Algarve, cujo trabalho motivou o estudo que realizei, expresso a minha gratidão pelo tempo que gentilmente me cedeu e pela experiência que partilhou, sem os quais me teria sido difícil compreender as aplicações com que trabalhei.

Ao Departamento de Informática, por me ter acolhido ao longo destes anos e por ter feito de mim uma pessoa melhor.

Resumo

Hoje em dia é possível encontrar pacotes de software em código aberto para resolução de problemas computacionalmente exigentes (HPC) que podem ser instalados com alguma simplicidade, até por utilizadores não-informáticos. Os desenvolvedores desses pacotes privilegiam, como seria de esperar, a portabilidade do seu software em detrimento de aspectos tais como "ter uma interface utilizador muito cuidada" ou "obter o máximo desempenho".

É neste último aspecto que focamos a nossa atenção: propomo-nos desenvolver uma metodologia simples que permita obter ganhos de desempenho satisfatórios em aplicações para as quais o código fonte está disponível, mas é de grande complexidade, não está documentado, não sendo, por isso, possível e/ou desejável modificá-lo.

O presente trabalho é o primeiro passo para verificar a viabilidade de tal propósito; exploram-se as possibilidades de optimização em três dimensões: arquitectura computacional, rede de interligação e software. Como "Caso de Estudo" aplicacional adopta-se o AMBER, um pacote de aplicações de Dinâmica Molecular, não deixando contudo de ensaiar uma outra aplicação similar, o NAMD. Exploram-se as três arquitecturas computacionais representativas do estado-da-arte dos sistemas ao alcance de uma pequena/média instituição de C&T: um cluster, um servidor cc-NUMA e, apenas para uma rápida comparação com as restantes, uma plataforma GPGPU.

Palavras-Chave: AMBER, Computação Paralela, Optimização.

Abstract

Today open source software packages that solve computationally demanding problems (HPC) are available, and may be installed even by non-expert users. Their developers, as one would expect, target software portability over aspects such as "nice user interface" or "reaching their topmost performance".

Performance is where we focus our attention: we aim to develop an simple methodology that leads to performance gains in applications for which source code is available, but are of great complexity, possibly undocumented and, therefore, it may be impractical to modify them.

The first step is a feasibility study, where we explore three optimization tracks: computational architecture, interconnection network, and software. As a "case study" we target AMBER, a Molecular Dynamics package; however, we also experiment with a similar application, NAMD. We explore the three state-of-the-art computing architectures that are available to small/medium R&T institutions: a cluster, a cc-NUMA server, and for a brief comparison with the other two, a GPGPU platform.

Keywords: AMBER, Parallel Computing, Optimization.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Problema e Objectivos

Hoje em dia é possível encontrar pacotes de software em código aberto para resolução de problemas computacionalmente exigentes (HPC) que podem ser instalados com alguma simplicidade, até por utilizadores não-informáticos. Os desenvolvedores desses pacotes privilegiam, como seria de esperar, a portabilidade do seu software sobre o maior número de arquitecturas, sistemas de operação, e middlewares eventualmente necessários, em detrimento de aspectos tais como "ter uma interface utilizador muito cuidada" ou "obter o máximo desempenho".

É neste último aspecto que focamos a nossa atenção: propomo-nos desenvolver uma metodologia simples, suportada num conjunto de heurísticas, e que permita obter ganhos de desempenho satisfatórios. A referida metodologia será suficientemente simples para poder ser usada por um utilizador não-informático, que já instalou e utiliza no seu dia-a-dia a aplicação, mas pode resultar em ganhos ainda maiores se aplicada, por exemplo, por um administrador de sistemas experiente. Em suma, tal metodologia poderá ser usada de forma incremental, de forma a obter ganhos de desempenho em aplicações para as quais o código fonte está disponível, mas é de grande complexidade, não está documentado, não sendo, por isso, possível e/ou desejável modificá-lo.

O presente trabalho é o primeiro passo para verificar a viabilidade de tal propósito; exploram-se as três dimensões principais do problema: arquitectura computacional (com especial relevo para os subsistemas de CPU e memória dos nós de computação), rede de interligação (procurando determinar a adequação da tecnologia e optimizar a sua utilização às necessidades da aplicação) e software (procurando as optimizações ao nível do conjunto de instruções do CPU, ao nível das bibliotecas e outro middleware, e ainda ao nível do sistema de operação).

Como "Caso de Estudo" aplicacional o presente trabalho explora fundamentalmente o AMBER, um pacote de aplicações de Dinâmica Molecular, não deixando contudo de ensaiar uma outra aplicação similar, o NAMD. Exploram-se as três arquitecturas computacionais representativas do estado-da-arte dos sistemas ao alcance (em termos financeiros) de uma pequena/média instituição de C&T: um cluster, um servidor cc-NUMA e, num breve ensaio com intuitos de mera comparação de desempenho, uma plataforma GPGPU.

Terminamos esta introdução destacando a necessidade de optimizar este tipo de aplicações, problema que nos foi colocado por uma Investigadora que usa o AMBER para fazer simulações de "Protein Folding", simulações estas que, no início deste trabalho, requeriam cerca de 1 ano de uso exclusivo dos 16 cores do cluster para produzir 4μs de simulação...

Capítulo 2

Introdução

Vivemos rodeados de informação.

Com a massificação das tecnologias de informação, os nossos hábitos alteraram-se profundamente. Alteraram-se também as metodologias. Nesse sentido, é justo dizer que as Tecnologias de Informação têm sido, e prevê-se que continuem a ser, um factor adjuvante da nossa recente evolução enquanto espécie inteligente.

Actualmente, um conjunto de áreas do conhecimento está dependente dos recursos computacionais que dispõe. Ciências, Engenharia, Medicina, e recentemente o ambiente empresarial, são alguns exemplos de realidades onde activamente são usados recursos computacionais para quebrar barreiras e ir mais além.

Mas há que não ignorar a pessoa comum, que representa todos nós — que já não sabemos viver sem um conjunto de tecnologias, e que esperamos ser continuamente surpreendidos; que desejamos obter o conhecimento para desenvolver terapêuticas que nos permitam combater doenças e prolongar a nossa existência; que queremos compreender melhor o Clima, bem como os fenómenos que nos rodeiam.

Muitos destes problemas são tão complexos, que é impraticável resolve-los num só computador. Em alguns, é tão importante obter a resposta, que desejamos atribuir o maior número de recursos disponíveis à tarefa em mãos, para encurtar o seu tempo de execução.

As nossas aspirações têm-se traduzido numa procura por tecnologias de baixo custo, de alta performance e elevada produtividade. São estas as forças que motivam a existência de arquitecturas computacionais de elevado desempenho. A taxonomia destas arquitecturas será exposta na secção 2.2. Por agora, importa reter que o uso de múltiplas unidades de processamento é inerente a estas arquitecturas, independentemente da sua concretização. Ao interligar um conjunto de processadores, possibilitamos o processamento paralelo – duas ou mais tarefas em execução simultânea. Como tal, o nosso objectivo é não só obter um poder computacional superior ao do melhor processador existente, como também conseguir uma relação custo-benefício mais favorável. Para além disso, tais sistemas podem oferecer tolerância a falhas (i.e., se um processador falhar, os restantes continuam disponíveis, não obstante a perda de performance). O seu enorme poder computacional permite-nos ambicionar resolver os grandes desafios computacionais com que somos confrontados. Numa altura em que a redução dos custos está a contribuir decisivamente para a expansão da computação de alta performance, a utilização de tais sistemas adivinha-se cada vez mais natural.

Concluindo, as arquitecturas multiprocessador têm transformado a forma como várias disciplinas da ciência e engenharia abordam, e resolvem, os seus problemas. Em grande medida, as tecnologias de informação, e em especial as arquitecturas de elevado desempenho, têm-nos permitido acelerar o ritmo ao qual adquirimos novo conhecimento, conservando recursos ao longo do caminho.

2.1 Aplicações

As arquitecturas multiprocessador têm sido utilizadas em diferentes áreas do conhecimento, o que nos tem permitido avanços extraordinários. Neste ponto procura-se mostrar algumas das aplicações destes sistemas.

Modelação e Simulação

Faz parte da natureza do Homem procurar conhecer o seu ambiente, compreendendo os diferentes fenómenos que o rodeiam. Questões como a modelação do Clima, simulações militares, e até mesmo a modelação da economia mundial, implicam a realização de simulações computacionalmente muito exigentes.

Alguns exemplos:

- Comportamento dos Mercados
- Meteorologia
- Oceanografia
- Sequenciação do genoma humano
- Simulação de semicondutores

Engenharia e Automação

Em muitos processos industriais a robótica e a automação têm sido usadas como forma de aumentar a produtividade, reduzindo o tempo de produção, o que tem permitido colocar novos produtos no mercado a um ritmo elevado. Por outro lado, a elevada competição entre empresas e economias torna vital a poupança de recursos. Nesse sentido, muitos procedimentos dispendiosos têm sido substituídos por simulações em computador.

Alguns exemplos:

- Aerodinâmica Computacional
- Inteligência Artificial
- Reconhecimento de Padrões
- Robótica

Energia

A energia é o motor de qualquer economia. Com o consumo mundial de energia a aumentar, em virtude da expansão das economias emergentes, é imperativo ampliar as fontes existentes, bem como procurar desenvolver novas fontes de energia. Numa área em que o sucesso não é garantido, os computadores permitem reduzir os custos associados às actividades de investigação e prospecção.

Alguns exemplos:

- Prospecção de gás e petróleo
- Estudos Geotécnicos
- Modelação de Reservatórios
- Energia Nuclear

Ciência e Investigação

A ciência está para a espécie humana, como a energia está para a economia. De facto, é a nossa sede de conhecimento que melhor nos define. Quase todas as áreas de investigação requerem o uso de computadores rápidos para prosseguir os seus estudos.

Alguns exemplos:

- Desenvolvimento de drogas
- Engenharia genética
- *Folding* de Proteínas
- Imagiologia

Cinema e Visualização

As arquitecturas de elevado desempenho têm sido aproveitadas para tornar a nossa vida melhor. O cinema, a animação e os jogos, são exemplos de realidades onde os meios computacionais são amplamente utilizados. Por isso mesmo, o cinema pode criar cenários incríveis, que desafiam a nossa imaginação; os programadores e *designers* podem desenvolver jogos e animações que fazem uso de GPUs *multicore*, apresentando um nível gráfico nunca antes experimentado.

Alguns exemplos:

- Visualização de Dados
- Realidade Aumentada

2.2 Arquitecturas Computacionais

Todas as aplicações apresentadas no ponto anterior têm em comum a sua natureza complexa. São tarefas exigentes e pouco triviais, que assentam em cálculos intensos onde, por ventura, participam inúmeros factores que têm necessariamente de ser observados; frequentemente, tais aplicações trabalham com um conjunto colossal de dados. Por estas razões, as arquitecturas sequenciais revelam-se incapazes de lidar em tempo útil com tamanho volume de processamento.

Procurando solucionar o problema, uma abordagem possível seria produzir componentes *hardware* suficientemente rápidos para a tarefa em questão. Tal aspiração cai por terra, tendo em conta a tecnologia disponível — actualmente encontramo-nos limitados pela velocidade da luz, pelas leis da termodinâmica e pelos elevados custos que enfrentaríamos ao produzir tal *hardware*. Por conseguinte, a alternativa viável tem consistido em distribuir o trabalho por um maior número de componentes.

Os processadores paralelos são sistemas compostos por múltiplas unidades de processamento, unidas por uma rede de interligação e pelo *software* necessário para permitir que tais unidades operem em conjunto. Fundamentalmente, estes sistemas caracterizam-se pela forma como as unidades de processamento comunicam – memória partilhada ou troca de mensagens – e pelo tipo de rede de interligação utilizada.

Antes de prosseguirmos com a descrição das arquitecturas relevantes, convém ter uma visão geral sobre a classificação das arquitecturas computacionais. A secção 2.2.1 apresenta o sistema de classificação de *Flynn*, proposto em 1966. A secção 2.2.2 progride a nossa discussão das arquitecturas de elevado desempenho.

2.2.1 Taxonomia de Flynn

A taxonomia de *Flynn* [1] é talvez o mais popular sistema de classificação das arquitecturas computacionais. A classificação baseia-se na noção de fluxos de informação: basicamente consideram-se duas dimensões distintas, instruções e dados, e o objectivo é classificar um sistema em função do processamento das instruções e dos dados. Comecemos por descrever os fluxos de informação: o fluxo de instruções define-se como a sequência de instruções executadas pela unidade de processamento; o fluxo de dados consiste na informação trocada entre a unidade de processamento e a memória. Segundo *Flynn*, cada um destes fluxos pode ser único ou múltiplo, o que dá origem às quatro categorias seguintes:

Single Instruction, Single Data (SISD) – o tradicional computador sequencial. Apenas um fluxo de instruções, e somente um fluxo de dados, tal como acontece no modelo proposto por John von Neumann.

Multiple Instruction, Single Data (MISD) – um único fluxo de dados alimenta múltiplas unidades de processamento, que operam independentemente umas das outras. Este é um modelo teórico, já que não se conhecem realizações comerciais do mesmo[1]. No fundo, o seu carácter "single data", restringe bastante o seu leque de aplicações.

Single Instruction, Multiple Data (SIMD) – todas as unidades de processamento executam a mesma instrução, podendo operar sobre diferentes fluxos de dados em simultâneo (*data parallelism*). Esta arquitectura é particularmente indicada para realizar operações regulares sobre um conjunto de dados. Por isso mesmo, é por exemplo indicada para aplicações de codificação de áudio e vídeo, e processamento de imagens. Quase todos os processadores actuais implementam, de alguma forma, um conjunto de instruções SIMD. A título de exemplo, os mais recentes processadores da Intel e AMD implementam as extensões SSE4 (*Streaming SIMD Extensions*).

Multiple Instruction, Multiple Data (MIMD) – o computador moderno. Uma máquina capaz de executar múltiplos fluxos de instruções (*control parallelism*), sendo que cada unidade de processamento pode trabalhar com distintos fluxos de dados. É o modelo que nos confere maior liberdade no que toca ao processamento paralelo. Por essa razão é correntemente a plataforma paralela mais comum.

Resta acrescentar que os computadores paralelos pertencem a uma das duas últimas classes (SIMD ou MIMD).

2.2.2 Arquitecturas de Memória Partilhada (*Shared Memory*)

Os sistemas de memória partilhada caracterizam-se pela existência de uma memória global, partilhada por todos os processadores. Os processos comunicam, coordenam-se e sincronizam-se lendo e escrevendo na memória. Como tal, o desenvolvimento de aplicações nestas plataformas encontra-se simplificado do ponto de vista do programador, ainda que seja sua responsabilidade assegurar que a existência de uma memória acessível por múltiplos processadores não invalida a correcção da aplicação.

A principal desvantagem destes sistemas é serem pouco escaláveis. Se múltiplos processadores tentam simultaneamente aceder à memória, pode ocorrer contenção, cuja expressão depende características da rede de interligação utilizada. Os sistemas mais comuns utilizam um único barramento central, o que restringe o número de processadores que acede em simultâneo ao bus. Por outro lado, os acessos à memória fazem-se, nos sistemas mais simples, um a um (podendo contudo ser *pipelined*). Ao adicionarmos mais processadores ao sistema, a contenção aumentará, o que afectará a performance.

Uma solução viável passa por usar caches; contudo, dispor de informação replicada por várias caches pode colocar-nos numa situação de incoerência entre caches; por conseguinte resta-nos garantir que a coerência é mantida. Tal tarefa requer *hardware* adicional, e inevitavelmente torna o sistema mais complexo. Porém, a exactidão das aplicações não mais depende do funcionamento da cache. Os sistemas de memória partilhada dividem-se em duas classes: UMA e NUMA.

[1] Não obstante, alguns autores [2] consideram que os processadores com um *multi-stage pipeline*, ou ainda os apelidados *systolic-array computers*, constituem exemplos de máquinas MISD [3].

2.2.2.1 *Uniform Memory Access* (UMA)

Nestes sistemas todos os processadores têm igual oportunidade de acesso à memória partilhada (mesma latência). A arquitectura SMP (*symmetric multiprocessor*), onde dois ou mais processadores idênticos comunicam entre si por via de uma memória partilhada, é o representante habitual desta classe. Os *multi-core* desktop actuais são outro exemplo; se cada núcleo for visto como um processador individual, então os *multi-core* são simplesmente SMPs encapsulados num único chip (*chip-level multiprocessing*).

2.2.2.2 *Nonuniform Memory Access* (NUMA)

Numa arquitectura NUMA, cada processador tem na sua proximidade parte da memória partilhada. O espaço de endereços é único, o que significa que cada processador vê todas as partes como um todo. Neste esquema, os tempos de acesso variam consoante a distância do processador à memória em questão, e dependem do *interconnect* utilizado. Se as caches são mantidas coerentes, a arquitectura diz-se cc-NUMA.

2.2.3 Arquitecturas de Memória Distribuída (*Distributed Memory*)

Ao contrário das arquitecturas de memória partilhada, as de memória distribuída são mais facilmente escaláveis. Estes sistemas são compostos por um conjunto de nós – uma unidade de processamento e memória local – que comunicam, enviando mensagens, através de uma rede de interligação.

Não existindo memória partilhada, não existe a necessidade de controlar os acessos à mesma. Tal pode reduzir a fracção de código que tem de ser executada em modo sequencial (regiões críticas), aumentado potencialmente o desempenho das aplicações paralelas. Por outro lado, o acesso a dados que estão noutros nós requer a troca de mensagens entre processos.

Uma grande desvantagem deste modelo é que exige do programador um esforço consideravelmente maior; de facto, torna-se indispensável coordenar a comunicação entre os nós (processos).

Ainda assim, por serem escaláveis, fáceis de construir e oferecerem uma boa relação custo-benefício, os sistemas de memória distribuída estão a dominar a área da computação paralela e de elevado desempenho.

2.2.4 *Clusters*

Os *clusters* são arquitecturas MIMD, logo paralelas e de memória distribuída, escaláveis e multi-propósito, concretizadas com componentes hardware prontamente disponíveis. Na prática, consistem em interligar um conjunto de nós (computadores), multiprocessador ou não, através de uma rede de ligação. Os *clusters* permitem-nos alcançar um elevado poder de computação, com um custo consideravelmente inferior à aquisição de um supercomputador equiparável. A gradual descida de preço dos componentes *hardware*, bem como a melhoria das redes de ligação e ainda o aparecimento de *middleware* de suporte, tornam estas arquitecturas particularmente atraentes, razão pela qual desfrutam, actualmente, de grande visibilidade.

Os *clusters* podem ser classificados segundo vários critérios, porém vulgarmente observa-se o tipo de utilização, classificando-os em "*Clusters* de Alto Desempenho" ou "*Clusters* de Elevada Disponibilidade". Os primeiros encontram-se em ambientes onde a computação domina e em que o objectivo típico não é fornecer um serviço, e.g. um *Web Server*, mas sim dispor de poder de computação suficiente para a realização de tarefas intensivas. Já os *clusters* de elevada disponibilidade reflectem a necessidade de eliminar pontos únicos de falha, o que se consegue através da redundância, e revela-se decisivo sempre que alguém depende do(s) serviço(s) que o *cluster* fornece.

2.2.4.1 Redes de Interligação

As redes de interligação assumem um papel decisivo no desempenho de uma arquitectura de memória distribuída: de pouco nos serve ter os melhores nós de computação, se utilizamos uma rede de fraco desempenho. No caso dos *clusters*, por exemplo, essas redes são, na prática, a essência do próprio *cluster*: sem elas, este não existiria.

Ao avaliarmos uma rede de interligação, frequentemente atendemos a três aspectos: a sua latência; a sua largura de banda; e a sua escalabilidade.

De seguida introduzem-se algumas tecnologias de interligação.

Ethernet

Esta é uma tecnologia amplamente utilizada nos mais diversos meios, oferecendo larguras de banda de 10, 100 e 1000 Mbps (respectivamente *Ethernet*, *Fast Ethernet*, *Gigabit Ethernet*); recentemente começaram a aparecer sistemas com interfaces a 10 Gbps.

Em ambientes IT, o padrão é o *Gigabit Ethernet*, já que o poder de computação e o volume de tráfego são elevados nestes ambientes. Actualmente, esta é também a tecnologia de interligação mais utilizada por alguns *clusters* de supercomputação, mas apenas nos de baixo custo, sendo, todavia, preterida nos mais rápidos supercomputadores do mundo[2] em benefício do *InfiniBand*.

As suas grandes vantagens são o custo de aquisição e manutenção relativamente baixo e o facto de ser uma tecnologia simples, estável e padronizada; como desvantagens, podemos referir a sua latência, que ronda as várias dezenas de μs e o consumo de CPU. No futuro espera-se a disponibilização de larguras de banda na ordem dos 100 Gbps.

Myrinet

Este é um *interconnect* proprietário (fornecido pela Myricom) de elevado desempenho, outrora muito usado. Esta tecnologia de comunicação baseada no envio de pacotes reduz o *overhead* ao nível dos protocolos quando comparada com as redes Ethernet, pelo que apresenta latências reduzidas e largura de banda elevada. A desvantagem é o seu custo quando comparado com a tecnologia Ethernet. A sua posição de destaque foi perdida para o *InfiniBand*.

InfiniBand

Esta tecnologia regista um crescimento anual de mercado na ordem dos 20%. A organização que mantém a especificação, a *InfiniBand Trade Association*[3], conta com participação de gigantes do mundo das IT, como a IBM, Intel e Oracle. A tecnologia é bidireccional, ponto-a-ponto e *multi-lane* (multi-pista).

Este *interconnect* está rapidamente a ganhar terreno ao *Gigabit Ethernet*. A razão prende-se com as suas larguras de banda muito elevadas, até 120 Gbps, e latências reduzidas (da ordem dos poucos μs), garantias ao nível da qualidade de serviço (QoS) e tolerância a falhas; e tudo isto com uma razoável relação custo-benefício.

2.2.5 Computação suportada em GPUs

A computação paralela chegou em grande. Ao enveredarmos por um caminho onde a evolução dos CPUs assenta, principalmente, na adição de mais *cores*, torna-se necessário que as nossas aplicações explorem a possibilidade de paralelismo; caso contrário não se verificarão grandes

[2] `http://www.top500.org/stats/list/36/connfam`
[3] `http://www.infinibandta.org/`

aumentos de desempenho ao migrar para sistemas mais recentes. Numa altura em que os grandes fabricantes anunciam processadores excedendo os 8 núcleos, começa-se também a olhar para outras formas de incrementar, ainda mais, o poder de processamento disponível. A tendência tem sido no sentido de aproveitar os recursos das *Graphics Processing Units* (GPUs).

Ao longo das duas últimas décadas, os processadores gráficos sofreram uma grande evolução desde a altura em que foram introduzidos no mercado de consumo; começaram pelos *pipelines* não programáveis (*fixed-function*) e são hoje chips programáveis capazes de um poder de processamento (GFLOPS) que envergonha qualquer CPU, já que possuem centenas de cores e memória com apreciável largura de banda. Por essa razão, não é de estranhar que estejam a ser construídos supercomputadores com este *hardware*. De facto, o actual supercomputador mais rápido do mundo possui mais de 7000 GPUs Nvidia Tesla, para além dos 14 mil processadores Intel Xeon, e atinge um máximo teórico de 4701 petaFLOPS[4].

Estes componentes têm uma grande vantagem em relação ao processador comum por exibirem um perfil energético consideravelmente mais favorável. Dado que a energia é um recurso escasso e dispendioso, este parece ser um argumento sólido. Contudo, foi necessário um processo de desenvolvimento até que se pudesse considerar utilizar GPUs para processamento multi-propósito, tal como se faz hoje em dia.

Actualmente a tecnologia encontra-se completamente operacional e em evolução; o maior desafio consiste em perceber até que ponto uma dada aplicação pode beneficiar deste tipo de computação, já que os GPUs são plataformas muito orientadas para cálculo de vírgula-flutuante, adequando-se melhor a problemas numéricos onde seja possível explorar o paralelismo dos dados. Por isso mesmo, as arquitecturas existentes consagram a utilização simultânea do CPU e do GPU.

2.3 Linguagens e Modelos de Programação Paralela

Nesta secção irão ser referidos alguns modelos e bibliotecas de suporte à computação paralela e de elevado desempenho. Para melhor compreender as possibilidades de cada uma das alternativas aqui estudadas, foram desenvolvidas pequenas aplicações (cujas listagens e medidas de desempenho podem ser consultadas nos Anexos A e B).

2.3.1 MPI

A troca de mensagens (*message-passing*) é um paradigma de programação, especialmente vocacionado para arquitecturas escaláveis de memória distribuída (e.g., *clusters*). Neste modelo, toda a interacção e partilha de dados acontece por troca explícita de mensagens.

O *Message Passing Interface* (MPI) é um *standard de facto*, concluído em 1994, que especifica um sistema de *message-passing* focado na eficiência, portabilidade e funcionalidade. Tal especificação baseia-se nas melhores práticas de sistemas semelhantes desenvolvidos ao longo dos anos que precederam o *standard*.

Um programa MPI consiste num conjunto de processos autónomos – com o seu próprio espaço de endereços – que comunicam entre si chamando as primitivas de comunicação. Estes processos pertencem a grupos, sendo univocamente identificados dentro do grupo pelo seu *rank* (um inteiro positivo).

Distinguem-se dois tipos de comunicação: ponto-a-ponto, onde um par de processos troca informação entre si, um enviando, o outro, recebendo; e colectiva, onde a informação é transmitida para todos os processos de um grupo. Em relação ao primeiro tipo, o MPI especifica rotinas de envio e recepção bloqueantes e não bloqueantes, o que nos permite optar entre um comportamento síncrono ou assíncrono. Já as rotinas de comunicação em

[4] `http://www.top500.org/system/10587`

grupo são de dois tipos: operações de comunicação, sendo o exemplo mais vezes mencionado o *broadcast*, onde um processo envia uma mensagem para todos os elementos do grupo; operações colectivas, como a soma, o máximo, o mínimo, entre outras funções a definir pelo utilizador. Existe ainda a possibilidade de sincronizar os processos através de uma barreira de sincronização (*barrier synchronization*).

As vantagens do uso de MPI são, nomeadamente, uma melhor compreensão das interacções e dos fluxos de informação entre os processos, um menor potencial para cometer erros de programação, bem como simplificar o *debug*e ainda escalável, como já o provou em *clusters* com milhares de nós.

Não obstante, e dependendo da complexidade da aplicação, poderá ser exigido um esforço considerável ao programador que pretenda paralelizar uma aplicação sequencial usando o MPI, uma vez que a ausência de uma memória partilhada poderá exigir inúmeros passos até que a coordenação necessária entre os processos seja atingida. Frequentemente, tal implica repartir um conjunto de dados, distribuindo-os pelos vários processos; adoptar um modelo de organização (e.g., *master/slave*[5]) e definir um protocolo de colaboração (i.e., quais as várias etapas de comunicação entre os processos).

2.3.2 OpenMP

O OpenMP é uma API (*Application Program Interface*) que permite o desenvolvimento de aplicações paralelas sobre memória partilhada. Consiste num conjunto de directivas de pré-processamento, que podem ser adicionadas ao código de um programa sequencial de forma a dividir o trabalho por um conjunto de *threads*.

O OpenMP apresenta um conjunto de vantagens apreciáveis: é simples, liberta o programador de ter de se preocupar com os detalhes, e permite paralelizar uma aplicação sequencial em muito pouco tempo, e com razoável esforço (especialmente se o código se basear em *loops*), já que frequentemente apenas são necessárias modificações mínimas ao código original.

Por outro lado, o OpenMP revela não ser escalável, devido ao *overhead* associado à gestão das *threads* (inicialização, escalonamento, criação e afectação de valores a variáveis privadas, ...). Por essa razão, a partir de um certo número de *threads* pode notar-se uma degradação do desempenho.

2.3.3 CUDA

Em finais de 2006, a Nvidia lançou no mercado o primeiro GPU construído sobre a arquitectura CUDA, a qual dispõe de recursos *hardware* adicionais que facilitam a computação multi-propósito. Designadamente, passou a ser possível utilizar a memória gráfica de forma natural – representando a possibilidade de definir estruturas de dados e de aceder a posições arbitrárias de memória – e recorrer a aritmética de vírgula-flutuante, dois aspectos essenciais e sem os quais se verificou ser bastante difícil utilizar os GPUs enquanto plataforma de computação genérica.

Para que fosse praticável a utilização de tais recursos sem ser necessário fazer uso das APIs DirectX ou OpenGL, a Nvidia criou a linguagem CUDA C, partindo da linguagem C *standard* e adicionando-lhe um número relativamente pequeno de *keywords* que permitem explorar as características inerentes à arquitectura.

Um programa CUDA é constituído por código executado pelo processador central, intitulado *host code*, e por código a ser executado pelo GPU, ou *device code*. Este último forma uma ou mais funções, os *kernels*, que irão ser chamadas pelo runtime. Uma das regras a observar é que o GPU apenas executa *device code*. Tal significa que, por vezes, bibliotecas

[5]O modelo *master/slave* determina a especialização de um dos processos, o qual terá um nível de responsabilidade acrescida em termos das operações que realiza (e.g., distribuição de trabalho, recepção dos resultados da computação de outros processos, I/O, etc.).

que o programador usa regularmente não existem na versão "device" e não podem, portanto, ser usadas.

Na nossa opinião, assente na breve experiência com uma plataforma CUDA, dois dos factores que actualmente podem travar a generalização desta plataforma são a sua aplicabilidade a domínios ou problemas que não envolvam cálculos vectoriais e a indisponibilidade de algumas bibliotecas na versão CUDA.

2.4 Bibliotecas para Cálculo Numérico

2.4.1 BLAS

BLAS (*Basic Linear Algebra Subprograms*) é uma especificação para um conjunto de rotinas para simplificar a resolução de problemas típicos de álgebra linear. Estão disponíveis implementações muito optimizadas, fornecidas pelos fabricantes de *hardware*, tais como a ACML (*AMD Core Math Library*), a Intel MKL (*Math Kernel Library*) e a CUBLAS da Nvidia.

As rotinas BLAS agrupam-se em três níveis distintos, segundo a sua funcionalidade: ***Level 1 routines***, que lidam com operações escalares e vectoriais (e.g., produto escalar entre dois vectores); ***Level 2 routines***, para operações entre matrizes e vectores; ***Level 3 routines***, para cálculos entre matrizes (e.g., multiplicação de matrizes).

Todas as rotinas apresentadas dispõem de uma versão de precisão simples e de uma versão de precisão dupla, suportando números reais e complexos.

O exemplo 2 (Apêndice A.2) mostra a realização da multiplicação de duas matrizes, usando a implementação da Nvidia. A função utilizada é a `cublasSgemm`, de nível 3, para números reais com precisão simples. Esta é, por ventura, uma das funções mais conhecidas da especificação.

Note-se que os *arrays* devem ser acedidos segundo a convenção *column-major*, adoptada pelo FORTRAN, sendo contrária à da linguagem C/C++. Tal significa que todos os elementos de uma coluna devem ser guardados em posições consecutivas e que as colunas devem ser guardadas pela sua ordem[6].

Os passos a dar para demonstrar a multiplicação de duas matrizes recorrendo à biblioteca CUBLAS são semelhantes àqueles necessários para executar uma aplicação CUDA genérica. Por conseguinte, é necessário reservar a quantidade de memória necessária, em ambos os contextos (*host* e *device*), inicializar as matrizes e transferi-las para a memória "gráfica"; chamar a função `cublasSgem`, que executará no GPU, e trazer o resultado para a memória do *host*.

2.4.2 LAPACK

LAPACK (*Linear Algebra PACKage*) é uma biblioteca, escrita em Fortran90, especializada em resolver problemas recorrentes de álgebra linear, como sistemas de equações lineares, mínimos quadrados lineares, decomposição de matrizes, entre outros. Na sua origem, em 1992, está a procura por tornar mais eficientes duas bibliotecas na altura já existentes, o EISPACK e o LINPACK. Tal foi alcançado, em grande medida, ao explorar as características das arquitecturas modernas, em particular as hierarquias de memória.

As rotinas LAPACK recorrem, tanto quando possível, à implementação BLAS disponível, e da qual dependem para obter ganhos de performance significativos, à semelhança de outros *packages* de álgebra linear.

As rotinas em LAPACK dividem-se em três grupos: ***driver routines***, que resolvem problemas concretos, bem definidos e que frequentemente envolvem um conjunto de passos

[6]Consultar `http://en.wikipedia.org/wiki/Row-major_order`

(e.g., um sistema de equações lineares); ***computational routines***, que realizam uma única tarefa (e.g., decomposição em valores singulares) e que poderão ser agrupadas para formar rotinas complexas; ***auxiliary routines***, que realizam tarefas computacionais de suporte, das mais diversificadas (e.g., cópia de *arrays*).

À semelhança das rotinas BLAS, em LAPACK especificam-se rotinas de precisão simples e dupla, para domínios reais ou complexos.

A grande vantagem do LAPACK assenta na exploração da estrutura dos algoritmos de álgebra linear, os quais tipicamente apresentam ciclos aninhados (*nested loops*), oferecendo um potencial de paralelismo notável. Por outro lado, questões pertinentes como a colocação (e permanência) de dados em memória são observadas.

Algumas das rotinas LAPACK podem ser encontradas em implementações optimizadas de bibliotecas matemáticas fornecidas pelos fabricantes de *hardware*.

2.5 Optimização

No quadro actual, as arquitecturas CISC (*Complex Instruction Set Computer*) são predominantes. A arquitectura Intel x86 (ou IA-32) é, sem dúvida, o membro mais popular desta família, ainda que em CPUs modernos as instruções x86 sejam traduzidas internamente em sequências de instruções simples, semelhantes às instruções RISC (*Reduced Instruction Set Computer*).

As arquitecturas modernas relevantes são superescalares, i.e., várias instruções podem ser executadas por ciclo (*instruction-level parallelism*) e exibem grande parecença entre si, muito simplesmente porque adoptam o modelo que tem provado funcionar mas também em consequência de princípios económicos básicos[7]). Em particular, as hierarquias de memórias nestas arquitecturas são muito semelhantes, diferenciando-se fundamentalmente ao nível das redes de interligação utilizadas.

Daqui se conclui que, numa arquitectura moderna, obter um melhor desempenho para uma aplicação sem efectuar alterações ao código fonte se consegue quase exclusivamente explorando as optimizações oferecidas pelo compilador, as hierarquias de memória e, eventualmente, o SO e o subsistema de I/O.

Nos pontos seguintes serão tecidas algumas considerações nesse sentido, procurando determinar como poderemos alcançar ganhos ao nível do desempenho, explorando soluções não invasivas[8].

2.5.1 Optimizações do Compilador

Os compiladores C/C++ frequentemente utilizados (*GNU Compiler Collection*, *Microsoft Visual C++*, etc.) suportam um conjunto de optimizações que, na maioria dos casos, possibilitam obter um melhor rendimento ao executar as nossas aplicações, sem grande variação no tempo de compilação. Regra geral, código mal estruturado ou pouco optimizado vê benefícios ao ser compilado com optimizações; contudo, na presença de um bom *design*, a probabilidade é que as optimizações não se traduzam em ganhos tão apreciáveis.

Concentrando a discussão no GCC, este suporta três grandes níveis de optimização, dois subníveis, e ainda um conjunto de optimizações que podem ser activadas individualmente. Na maior parte dos casos, quando não existe uma ideia clara do que se pretende que o compilador faça, compilaremos fazendo:

```
gcc -On foo.c ,
```

[7] Nenhuma empresa quer estar atrás dos seus concorrentes.
[8] Que não implicam alteração directa de código.

onde `n` é o nível de optimização. Podemos activar um dos seguintes níveis: 1, 2, ou 3. Do nível 1 para o 3, cresce a complexidade das optimizações, assim como pode crescer o número de instruções máquina contidas no executável produzido[9].

Em concreto, o nível 1 (O1) procura produzir código optimizado sem despender muito tempo a fazê-lo. Por essa razão, apenas as optimizações mais triviais são activadas (e.g. não se faz reordenação de instruções). Passando para o nível 2 (O2), são activadas as mesmas optimizações que em O1 e ainda um conjunto de outras que, contudo, não levam ao aumento do tamanho do executável. Já o nível 3 (O3) acrescenta ao nível 2 um conjunto de optimizações que, podendo aumentar ainda mais a performance, tem como desvantagem o aumento do tamanho do executável (e.g., *loop-unroling*[10], funções *inline*). Tal pode, porém, levar a uma perda de performance em última análise, pois como sabemos a cache de instruções do CPU é limitada.

Diz a experiência que a opção pelo nível dois é a que mais frequentemente produz o melhor resultado (melhor compromisso entre desempenho e tamanho). Porém, sempre que viável, deve-se recorrer à experimentação.

Em simultâneo com a activação de um nível de optimização, é possível especificar qual o CPU alvo para o nosso executável. Por omissão, o GCC gera código contendo instruções capazes de serem executadas por qualquer plataforma x86. Tal é bem-vindo sempre que a portabilidade seja algo desejável. Contudo, se sabemos que a nossa aplicação vai correr exclusivamente numa máquina em particular, é preferível usar a opção `-march=native`, que autoriza o compilador a utilizar instruções unicamente suportadas pelo CPU em causa, o que poderá aumentar o desempenho.

Uma ampla descrição de cada uma das optimizações activadas em cada nível pode ser consultada no manual do GCC[18]. A sua introdução sai fora do âmbito desta exposição, ainda que este assunto possa vir a ser retomado mais tarde.

Recuperando novamente o já aludido exemplo da multiplicação de matrizes, foram executados alguns testes de forma a verificar o comportamento das optimizações supramencionadas. A tabela 2.1 apresenta os resultados.

Tabela 2.1: Tempos de execução para o exemplo 1 (Apêndice A.1) quando compilado nas configurações apresentadas.

	Tamanho (bytes)	Tempo (s)	Speedup (×)
Sem optimização	2484	8,898	---
-O1	2422	4,478	1,99
-O2	2406	4,493	1,98
-O3	2566	4,436	2,01
-march=native	2484	8,945	0,99
-O2 -march=native	2406	4,479	1,99

Como se pode observar, o nível 3 de optimização aumenta o tamanho do executável[11], conforme anteriormente se indicou. Não obstante, é neste nível que se obtém o melhor *speedup*[12], ainda que por curta margem. Note-se que o nível O2 produz o executável com

[9] Em Linux tal vê-se consultando o tamanho da secção *text* de um executável (ELF).

[10] Corresponde a alterar um ciclo (*loop*) de forma que em cada iteração seja realizado maior trabalho.

[11] Embora tal seja verdade, os valores apresentados na tabela dizem respeito ao tamanho da secção *text* do executável.

[12] Rácio entre o tempo de execução de um algoritmo sequencial e o tempo de execução desse mesmo algoritmo quando executado em paralelo. Ao longo deste trabalho usaremos também a palavra *speedup* para quantificar a melhoria introduzida pela recompilação de determinada aplicação, de acordo com um conjunto de parâmetros a indicar.

menor número de bytes, sendo aproximadamente 7% menor do que o executável produzido pelo terceiro nível de optimização. Constata-se ainda que neste caso a geração de instruções específicas para a máquina em questão se relevou vã quando isolada, ainda que de forma muito ténue.

Concluindo, reitera-se a importância de realizar experimentação, sempre que possível, pois o resultado é muito dependente do código da nossa aplicação.

No ponto seguinte é introduzido um tipo muito específico de optimização, que faz uso de características dos CPUs modernos.

2.5.2 Instruções SSE

Como foi referido no ponto 2.2.1, os processadores actuais implementam um conjunto de instruções SIMD. Ao incluirem uma arquitectura vectorial, estes processadores oferecem ganhos de desempenho apreciáveis em aplicações que executam frequentemente a mesma operação sobre um conjunto de dados tal, que pequenos subconjuntos seus possam ser tratados individualmente.

A tecnologia SSE (*Streaming SIMD Extensions*), sucessora do MMX, foi introduzida em 1999 nos processadores Intel Pentium III (IA-32) e compreende um conjunto de 70 instruções de aritmética, lógica e outras, que dispõem de 8 registos de 128 bits (`xmm0, xmm1, ..., xmm7`), cada um destes aptos a acolher 4 números em vírgula-flutuante de precisão simples (32 bits)[13].

Para melhor se compreender a pertinência das extensões SIMD, nada melhor do que um breve exemplo. Admita-se que se pretende somar dois vectores. Para facilitar, será fixado em 4 o número de elementos de cada vector, os quais serão números em vírgula-flutuante de precisão simples. Em linguagem C, a função mais simples para calcular o pretendido seria:

```
void f (float *m, float *n, float *p)
{
   p[0] = m[0] + n[0];
   p[1] = m[1] + n[1];
   p[2] = m[2] + n[2];
   p[3] = m[3] + n[3];
}
```

Esta função, quando compilada pelo GCC sem activar as extensões SSE, corresponde ao seguinte código *assembly*:

```
f:
   push  ebp
   mov   ebp, esp
   mov   ecx, DWORD PTR [ebp+8]
   mov   edx, DWORD PTR [ebp+12]
   mov   eax, DWORD PTR [ebp+16]
   fld   DWORD PTR [ecx]
   fadd  DWORD PTR [edx]
   fstp  DWORD PTR [eax]
   fld   DWORD PTR [ecx+4]
   fadd  DWORD PTR [edx+4]
   fstp  DWORD PTR [eax+4]
   fld   DWORD PTR [ecx+8]
   fadd  DWORD PTR [edx+8]
```

[13] Versões mais recentes da tecnologia surgiram entretanto, as quais já suportam dupla precisão. Neste capítulo introdutório não é nosso objectivo introduzir todas as versões existentes e suas diferenças, mas sim caracterizar a essência da tecnologia e demonstrar como ela pode ser útil no contexto de uma optimização não invasiva.

```
   fstp  DWORD PTR [eax+8]
   fld   DWORD PTR [ecx+12]
   fadd  DWORD PTR [edx+12]
   fstp  DWORD PTR [eax+12]
   pop   ebp
   ret
```

Antes de analisarmos o que aconteceria se a função `f` fosse compilada activando o uso das extensões SSE, contemplemos primeiro a função `fsse`, escrita de forma a indicar explicitamente ao compilador qual a instrução SSE que pretendemos que seja utilizada. Por simplicidade, optou-se por utilizar uma das funções intrínsecas que o GCC disponibiliza para o efeito (em vez de escrever directamente código *assembly* na fonte C):

```
void fsse (__m128 * m, __m128 * n, __m128 * p){
   *p = _mm_add_ps (*m, *n);
}
```

Para compilar este novo código em GCC devemos executar:

```
gcc -mfpmath=sse -msse foo.c
```

A chamada `_m_add_ps` fará com que o compilador gere código *assembly* que em dado momento executará a instrução `addps`, a qual aceita dois argumentos correspondentes a dois registos SSE. Concretamente,

```
addps xmm1, xmm2    ,
```

coloca no registo `xmm1` o valor da soma de `xmm1` com `xmm2`. A figura seguinte ilustra a operação descrita.

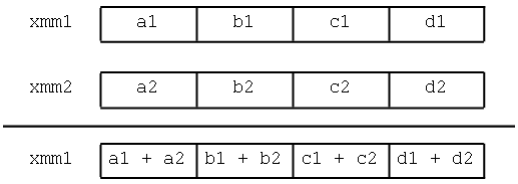


Figura 2.1: Exemplo do funcionamento da instrução `ADDPS`.

Para a função `fsse`, o compilador gera o seguinte código *assembly*:

```
fsse:
   push   ebp
   mov    ebp, esp
   sub    esp, 40
   mov    eax, DWORD PTR [ebp+12]
   movaps xmm0, XMMWORD PTR [eax]
   mov    eax, DWORD PTR [ebp+8]
   movaps xmm1, XMMWORD PTR [eax]
   movaps XMMWORD PTR [ebp-24], xmm1
   movaps XMMWORD PTR [ebp-40], xmm0
   movaps xmm0, XMMWORD PTR [ebp-40]
   movaps xmm1, XMMWORD PTR [ebp-24]
```

```
addps  xmm0, xmm1
mov    eax, DWORD PTR [ebp+16]
movaps XMMWORD PTR [eax], xmm0
leave
ret
```

As instruções SSE são apresentadas em negrito. Como esperado, perto do fim vemos a instrução `addps`, introduzida previamente. Em número de instruções, o código agora apresentado é apenas 3 instruções mais curto que o seu homólogo. Ainda assim, e não obstante este ser um programa extremamente trivial, a versão `fsse` demora aproximadamente metade do tempo a executar do que a versão `f`. Tal corresponde a um speedup de $2\times$, equivalente a dobrar o número de processadores de uma máquina, o que é bastante significativo. Daqui se depreende a pertinência das extensões SIMD.

Mas o que aconteceria se a função `f` tivesse sido compilada activando as extensões SSE? O seguinte código *assembly* dá-nos a resposta:

```
f:
   push  ebp
   mov   ebp, esp
   mov   ecx, DWORD PTR [ebp+8]
   mov   edx, DWORD PTR [ebp+12]
   mov   eax, DWORD PTR [ebp+16]
   movss xmm0, DWORD PTR [ecx]
   addss xmm0, DWORD PTR [edx]
   movss DWORD PTR [eax], xmm0
   movss xmm0, DWORD PTR [ecx+4]
   addss xmm0, DWORD PTR [edx+4]
   movss DWORD PTR [eax+4], xmm0
   movss xmm0, DWORD PTR [ecx+8]
   addss xmm0, DWORD PTR [edx+8]
   movss DWORD PTR [eax+8], xmm0
   movss xmm0, DWORD PTR [ecx+12]
   addss xmm0, DWORD PTR [edx+12]
   movss DWORD PTR [eax+12], xmm0
   pop   ebp
   ret
```

As instruções em negrito denunciam o uso de instruções SSE. No entanto, o código produzido está longe de ser o desejado. Na verdade, o esquema é muito semelhante ao código *assembly* gerado pela mesma função, quando compilada sem as extensões SSE. Tal deve-se ao facto de o compilador usar a instrução escalar `addss`, a qual — ao contrário da anteriormente descrita — apenas trabalha sobre 32 dos 128 bits dos registos `xmm`, o que explica porque vislumbramos essa instrução quatro vezes. Concluindo, irão ser executadas instruções SSE mas o programa continuará a seguir a lógica SISD.

A tabela seguinte permite melhor conhecer os resultados obtidos em várias configurações. Note-se que a versão `fsse` é sempre compilada, em qualquer dos casos, activando as instruções SSE.

Tabela 2.2: Tempos de execução comparando o desempenho da versão `f` com a versão `fsse`.

	Versão `f` (ns)	Versão `fsse` (ns)	Speedup (×)
Sem optimização	1300	664	1,96
-O1	1295	1318	0,96
-O2	1301	1299	1,00
-O3	646	1321	0,49
-march=native	1268	671	1,89
-mfpmath=sse -msse	1331	---	---

Os resultados obtidos complementam o anterior.

Conclui-se que o comportamento obtido ao compilar uma dada aplicação activando as instruções SSE, sem contudo alterar o código, é algo que tem de ser investigado caso a caso. Relatos da comunidade parecem indicar ser possível obter ganhos modestos em aplicações não preparadas, ainda que o código gerado não seja, por ventura, o melhor[14]. Ainda, poderá dar-se o caso de diferentes compiladores terem distintas capacidades de gerar instruções SSE a partir de código genérico como o da função anteriormente designada de inicial (e.g., o *Intel C/C++ Compiler*). Neste aspecto, e pelo potencial que exibem, as extensões SSE são algo a explorar.

2.5.3 Caches

A cache é um tipo de memória inerentemente rápida, utilizada para colocar na proximidade do CPU informação frequentemente utilizada. Na hierarquia de memórias ocupa o segundo nível, sendo que primeiro aparecem os registos do CPU, os quais se considera serem acedidos sem qualquer latência (0 ciclos).

Os sistemas actuais dispõem tipicamente de dois níveis de caches[15], L1 e L2, os quais permitem minimizar, dentro do possível, os acessos a memória mais lenta, nomeadamente à memória RAM.

Quando uma palavra[16] não é encontrada na cache L1, ela tem de ser transferida do nível inferior, que se espera ser a cache L2, mais lenta mas com maior capacidade. Diz-se que ocorre um "*cache miss*" quando um processo tem de aguardar vários ciclos por uma palavra não se encontrar na cache L1. A tabela seguinte apresenta as características[17] principais das caches da máquina onde têm sido executadas as aplicações cujo código fonte apresentamos no Apêndice A.

[14] O que acontece de facto é que a capacidade de vectorização dos compiladores é limitada. Por essa razão a versão 4.1 das extensões já inclui instruções que ajudam nesse sentido.

[15] Os novos processadores com quatro ou mais cores já dispõem de três níveis de caches (L1, L2 e L3). Para facilitar a discussão, tal será deliberadamente ignorado, sem compromisso da validade dos conceitos a apresentar.

[16] Um conjunto fixo de bits, que é tratado como um todo.

[17] Estes e outros valores apresentados para caracterizar as memórias do sistema considerado foram medidos com um *software* de *benchmark*. Consultar `http://www.cpuid.com/softwares/pc-wizard.html`.

Tabela 2.3: Características generais das caches disponível na máquina de testes. Ambas as caches são associativas por grupos.

	L1	L2
Tamanho (KBytes)	2×32	6144
Linha (bytes)	64	64
Esquema	8-way	24-way
Latência (ciclos)	3	14

Sabendo que o tempo de acesso à memória cresce à medida que se desce na hierarquia, então a penalização é imposta pelo facto de uma palavra não se encontrar na cache L2 é consideravelmente superior àquela que se coloca quando não é possível obter uma palavra da cache L1. Em concreto, havendo um *miss* na cache L2, teremos de aceder à memória RAM, cuja latência fica perto de 200 ciclos no sistema considerado. Por oposição, se for possível obter a palavra pretendida de L2, enfrentaremos um custo dado pela soma das latências de L1 e L2, que não chega a 10% do valor anterior.

Devido à penalização incorrida ao não encontrar a palavra desejada em L2, os acessos à memória RAM (ou pior, a níveis inferiores) dominam o custo da computação. Uma boa utilização das caches revela-se fundamental para minimizar a distância[18] que separa o CPU e a memória intermédia.

Antes de prosseguir, convém contudo recordar como a memória é gerida pelo sistema de operação.

Existem dois espaços de endereços distintos nas arquitecturas modernas. Os endereços gerados pelo CPU, denominados lógicos, e os endereços gerados pela MMU (*Memory Management Unit*), designados endereços físicos.

Tendo isso em reflexão, todos os sistemas de operação de interesse suportam uma técnica de gestão de memória denominada paginação. Essencialmente, a paginação consiste em repartir a memória física em blocos de dimensão fixa, ou *frames*, e a memória lógica em porções da mesma dimensão, as chamadas páginas. Tal dimensão é definida pelo *hardware* e é, tipicamente, uma potência de 2.

A cada processo é-lhe atribuído um conjunto de páginas, correspondendo a *frames* não necessariamente contíguas da memória física. Neste contexto, os endereços lógicos gerados pelo CPU são formados por duas partes: um número de página e um *offset* (deslocamento dentro da página). A tradução de um endereço lógico para o correspondente endereço físico faz-se consultando uma tabela de páginas, a qual indica para cada página qual o endereço base na memória física.

Cada processo tem a sua própria tabela de páginas, a qual reside em memória principal e é apontada pelo registo PTBR (*Page Table Base Register*). Sendo assim, quando o processo pretende aceder a um elemento de uma página, tem de consultar a tabela de páginas em memória para obter o endereço físico do elemento, e posteriormente aceder de novo à memória para obter esse mesmo elemento. Tal perfaz dois acessos à memória. Isto significa que perto de 400 ciclos serão utilizados no acesso, o que não é de todo aceitável.

A solução é usar uma cache rápida e especializada (e por isso dispendiosa), tipicamente com um número reduzido de entradas, a que se dá o nome de TLB (*translation look-aside buffer*).

[18] Significando latência.

Esta cache conterá apenas algumas das entradas presentes na tabela de páginas. A ideia é eliminar a necessidade de aceder à memória principal para consultar a tabela de páginas. Dessa forma, quando a TLB contém a entrada desejada, basta apenas um único acesso à memória principal para obter o elemento em questão, o que é deveras vantajoso.

Quando esta cache não dispõe de informação para a página que se pretende, o processo segue os mesmos contornos tal como descrito em cima. Neste caso, dizemos que ocorreu um *TLB miss* e incorremos na penalização dos 400 ciclos.

Posto isto, pode ser interessante explorar a TLB; o objectivo é assumidamente reduzir a probabilidade de uma referência para uma página não estar na TLB. Idealmente gostaríamos que o número de páginas de um processo não excedesse o número de entradas na TLB, pois tal reduziria a probabilidade de insucesso ao consultar a TLB. No entanto, com páginas de 4 KBytes e um número pequeno de entradas isso só é possível se a aplicação tiver um espaço de endereços diminuto.

A única solução passa por utilizar páginas consideravelmente maiores.

Em arquitecturas x86-64 com Linux é possível utilizar páginas de 4MB, conhecidas por *hugepages*; e se recorrermos à biblioteca *libhugetlbfs* [8] não nos é necessário recompilar o código da aplicação. Para tal, apenas necessitamos de um *kernel* com o adequado suporte.

2.5.4 Ligação Dinâmica vs. Ligação Estática

Um dos princípios mais vincados da engenharia de *software* assenta na reutilização de *software*, o que implicitamente pressupõe uma separação de tarefas, que em última análise nos leva à especialização ou modularização. De facto, a dinâmica da área vive de construir recorrendo ao que já existe, até porque caso contrário seria impraticável acompanhar o ritmo do mundo moderno.

Por essa razão todo o programador está familiarizado com o conceito de biblioteca. As bibliotecas são código compilado de forma a poder ser utilizado por outras aplicações.

Nos sistemas de operação esse código existe no sistema de ficheiros. Particularmente, nos ambientes UNIX a directoria `/lib` aloja um conjunto de ficheiros de extensão "so" (*shared objects*) que correspondem às bibliotecas partilhadas, as quais são requeridas por um conjunto de aplicações. Quando assim é, dizemos que estas bibliotecas foram ligadas dinamicamente ao código das respectivas aplicações. Tal significa que o executável não contém no seu interior o código da(s) biblioteca(s) que usa, pelo que quando necessário esse mesmo código deve ser trazido para memória, se ainda não lá estiver.

Contrariamente a este processo existe a ligação estática, que determina que o código de uma ou mais bibliotecas seja copiado para o executável da aplicação.

Comparando as duas abordagens, existem vantagens e desvantagens como seria expectável. No que diz respeito à ligação estática, assumidamente o tamanho do executável será maior. Ainda, perde-se flexibilidade já que, por exemplo, uma pequena mudança ou correcção de uma falha numa biblioteca implica ter de recompilar todo o código, o que em caso de um programa de grande dimensão pode ser moroso. Por outro lado, se existirem várias cópias do executável em memória, então estaremos a desperdiçar recursos vitais.

No entanto, pode-se incorrer em benefício ao não ser necessário localizar a biblioteca e carregá-la em memória, como acontece no caso da ligação dinâmica[19]. Não obstante, após essa biblioteca ser carregada, vamos assumir, do disco, o comportamento será idêntico à ligação dinâmica, a não ser que por alguma razão o sistema de operação retire essa biblioteca

[19]Na ligação dinâmica devemos ter ainda em consideração o preenchimento da *Procedure Linkage Table* (PLT) com os endereços onde as funções da biblioteca estão carregadas em memória. Mesmo após esse preenchimento estar completo, a tabela continua a ser consultada cada vez que se pretende chamar uma função de uma biblioteca ligada dinamicamente.

da memória principal (tal pode acontecer quando, por exemplo, o SO é confrontado com uma situação de pouca memória e tem de escolher as páginas que serão enviadas para disco, através de um algoritmo de substituição de páginas, num processo denominado *page-out/in*). Em Linux a ligação dinâmica é a regra.

2.5.5 *InfiniBand*

Nesta secção será aprofundada a breve introdução feita à tecnologia *InfiniBand* (IB). Antes disso, impõe-se justificar o assumido destaque dado à tecnologia referida. A razão é simples: o IB é conhecido por reduzir os *overheads* próprios dos protocolos de comunicação tradicionais. Como tal, pode-se dar o caso de uma aplicação vir a beneficiar com a adopção do *InfiniBand* para a arquitectura onde esta executa, em prejuízo da tecnologia já instalada (e.g., *Gigabit Ethernet*).

O *InfiniBand* define uma rede de comunicação (*IBA fabric*) confiável, de elevada largura de banda e reduzida latência, que interliga nós (computadores, *routers* e sistemas de I/O) através de um conjunto de *switches*.

Na maioria dos casos, a cada um dos nós representados corresponderá uma máquina independente, possivelmente heterogénea, onde executam um ou mais processos, denominados consumidores pela especificação IB.

A troca de dados assenta na existência de filas (*work queues*) concretizadas ao nível do *channel adapter*, onde são colocadas instruções a serem executadas. Estas filas são sempre criadas em número par, numa associação a que a especificação designa de *Queue Pair* (QP) – uma fila para envio (*send*) e outra para recepção (*receive*). Nessa lógica, a cada processo corresponderá um ou mais *Queue Pairs*, independentes dos demais.

A comunicação faz-se, sucintamente, conforme se segue. Um consumidor submete um pedido de trabalho, o que implicará a colocação de um *Work Queue Element* (WQE) na *work queue* adequada, para posterior tratamento pelo *channel adapter*, segundo a disciplina FIFO. Completado um WQE, o *channel adapter* coloca um *Completion Queue Element* (CQE) numa fila terminal (*completion queue*), associada à correspondente *work queue*. Ambos os elementos mencionados, WQE e CQE, contêm a informação necessária para realizar as operações solicitadas.

No que se refere ao envio de informação, estão disponíveis quatro operações:

SEND – um WQE, especificando um bloco de dados existente no espaço de memória do utilizador, é colocado numa *send queue*; o *hardware* envia os dados ao receptor, sendo este o responsável por determinar onde tais dados serão colocados.

RDMA-WRITE – semelhante à operação **SEND**, mas com a particularidade que o WQE especifica um endereço de memória remoto[20] onde serão colocados os dados enviados. Nesta operação, e nas restantes operações do género, está implícita uma forma de acesso directo a memória remota.

RDMA-READ – o WQE especifica o endereço remoto que será utilizado pelo *hardware* para transferir dados da memória remota para a memória do consumidor.

ATOMIC – possibilita duas acções distintas e atómicas sobre blocos de memória remotos de 64-bit.

[20] A memória remota pode ser, na verdade, um *Queue Pair* de um outro processo na mesma máquina que o emissor.

- "*Compare and Swap if Equal*" – lê um valor v1 da memória remota e compara-o com um valor v2 definido pelo consumir, substituindo v1 na memória remota por um valor v3, também indicado pelo consumidor, sempre que v1 seja igual a v2;
- "*FetchAdd*" – lê um valor v1 da memória remota, entrega-o ao QP requisitante, e substitui v1 na memória remota por um valor v1 + v2, tendo o requisitante determinado v2.

Para receber, está disponível uma única operação, *Post Receive Buffer*, cujo propósito é designar um *buffer* local onde serão colocados os dados a chegar.

As operações aqui apresentadas permitem vislumbrar as razões pelas quais a tecnologia *InfiniBand* é verdadeiramente inteligente. De facto, ao contrário de protocolos de rede complexos, como o TCP/IP, que envolvem muito código a ser executado no contexto do *kernel*, bem como cópias de *buffers* do *user space* para o *kernel space*, que ocupam o CPU, o *InfiniBand* trabalha com dados residentes no espaço de memória do utilizador, sendo que esses mesmos dados são processados e movidos pelo *channel adapter*, o que implica retirar o CPU e o *kernel* da equação.

Dessa forma, preciosos ciclos de processador podem ser preservados, o que certamente fará a diferença para aplicações que, correndo em arquitecturas paralelas como os *clusters*, trocam muita informação entre os seus nós. Desde já se antecipa o potencial de tal tecnologia para suportar aplicações baseadas em MPI.

Capítulo 3

Caracterização das Infra-estruturas

Desenvolveremos um conjunto de estudos, procurando transpor para "o mundo real" as ideias e técnicas introduzidas previamente, auxiliados por três arquitecturas distintas que dispomos: um *cluster* homogéneo de 4 nós; um SMP (cc-NUMA) com 16 *cores*; e por último uma plataforma GPGPU de última geração.

O propósito deste capítulo consiste em enunciar as especificações das máquinas que utilizaremos e, adicionalmente, em quantificar o seu desempenho, facilitando futuras comparações. Para caracterizar as infra-estruturas de rede usamos dois níveis de testes: o primeiro, no qual utilizamos os *benchmarks* `netperf` e `perftest`[1], caracteriza as infra-estruturas ao nível do protocolo de transporte; o segundo, no qual utilizamos o *benchmark* `mpitests` caracteriza as infra-estruturas do ponto de vista das aplicações MPI. Para medir o desempenho do subsistema de memória usamos o *benchmark* STREAM.

3.1 *Cluster* – HP ProLiant DL145 G2

Cada nó do *cluster* é um servidor HP ProLiant DL145 G2, sistema *entry-level*, *rack-mount* (1U), baseado na plataforma AMD Opteron 200 Series, que foi lançada no mercado entre 2005 e 2006.

Eis as principais especificações de um HP DL145 G2:

- 2× Dual-Core AMD Opteron 275 (Italy), 2.2GHz, 90nm;
- 4 GB DDR PC3200 (400MHz), Advanced ECC;
- 2× Maxtor 6L080M0 80GB, 7200 RPM, 8MB Cache, SATA 1.5Gb/s;
- Integrated Dual Broadcom 5721 Gigabit Ethernet NICs;
- Mellanox Technologies MT23108 InfiniHost, 10 Gbps dual-port.

Relembrando a discussão do capítulo precedente, é do nosso interesse conhecer as características das caches para o sistema agora descrito. Para tal utilizámos a aplicação `cpuid`[2], a qual fornece inúmeras informações[3] sobre o (s) processador (s) instalado (s).

Constatámos, então, que o processador em causa dispõe de dois níveis de caches, L1 e L2, sendo que é imposta a distinção entre cache de dados (*data* cache) e cache de instruções (*instruction* cache) no nível superior (L1), contrariamente ao que acontece no nível inferior (L2)[4].

[1] *OpenFabrics Enterprise Distribution*

[2] Consultar o manual em /usr/share/man/man1/cpuid.1.gz para maior detalhe.

[3] Revelou-se imprescindível consultar o manual "*BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*" disponível em `http://support.amd.com/us/Processor_TechDocs/26094.PDF`.

[4] Dizemos que a cache L2 é uma cache unificada.

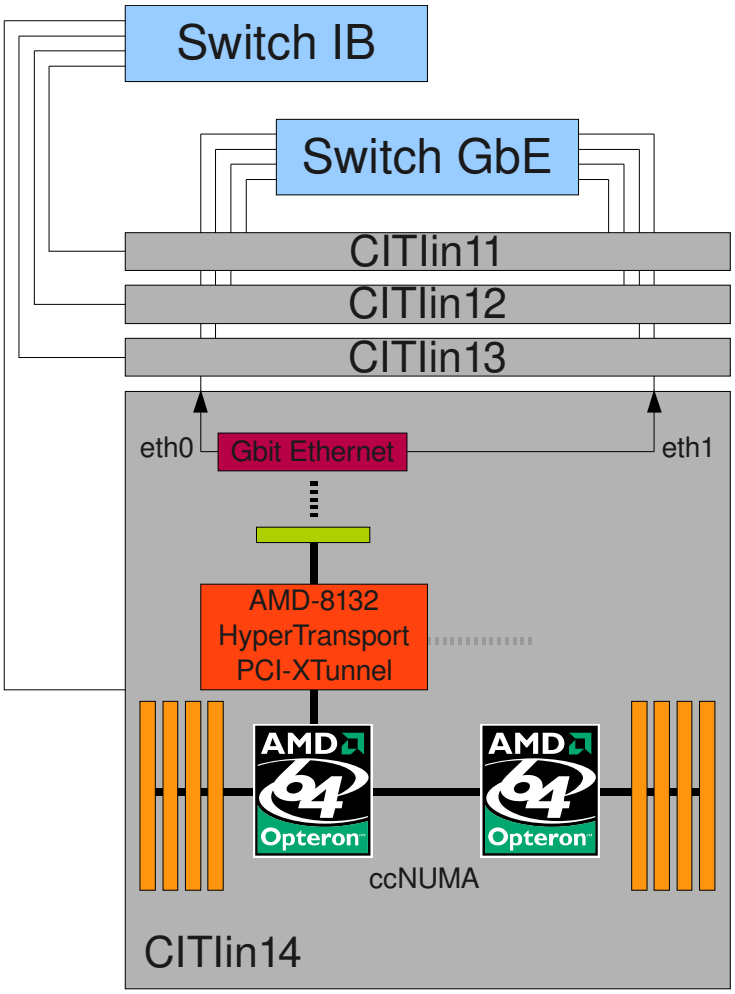


Figura 3.1: Representação do *cluster*. A ligação entre os processadores faz-se através de *HyperTransport* a 1GHz. Sendo uma arquitectura cc-NUMA, os tempos de acesso à memória (representada a amarelo torrado) não são uniformes, dependendo da distância à memória em questão.

Em relação à TLB, somos informados da existência de duas caches distintas, concretizando também aqui uma hierarquia de dois níveis. Tendo em conta os valores que recolhemos, somos levados a concluir que a TLB de nível inferior apenas suporta as convencionais páginas de 4K, o que poderá ter implicações ao nível da utilização de *hugepages*.

As tabelas 3.1 e 3.2 completam a visão que actualmente temos das caches do sistema[5]. Prosseguimos agora com os resultados dos *benchmarks* referidos.

3.1.1 *Benchmarks* da Infra-estrutura de Rede

Começaremos por utilizar os *benchmarks* ao nível de transporte e terminaremos realizando um *benchmark* que efectua o envio e recepção de mensagens MPI — complementando a visão que então dispusermos. Para o efeito usaremos apenas dois nós do cluster, considerando quatro opções: *InfiniBand* (IB) – denominado nativo – *IP over InfiniBand* (IPoIB), *IP over Ethernet* (IPoE) e *Myrinet Express over Ethernet* (MXoE)[6].

[5]Gostaríamos de poder apresentar as latências de tais caches, porém desconhecemos uma aplicação capaz de o fazer em ambiente Linux.

[6]Dispomos da última opção por termos instalado este protocolo de baixa latência disponível em `http://open-mx.gforge.inria.fr/`.

Tabela 3.1: TLB.

Cache	Tamanho da página	# Entradas		Esquema
		Instruções	*Data*	
L1 TLB	2M	8	8	Associativa pura
	4K	32	32	Associativa pura
L2 TLB	2M	---	---	---
	4K	512	512	4-way

Tabela 3.2: L1 e L2 *data* & *instruction*.

Cache	Tamanho (KBytes)	Linha (bytes)	Esquema
L1 data cache	64	64	2-way
L1 instruction cache	64	64	2-way
L2 unified cache	1024	64	16-way

Para que possamos apreciar os seus resultados em pleno, devemos introduzir o conceito do *Interrupt Coalescing*. Como sabemos, as interrupções são a forma do *hardware* sinalizar ao SO (via CPU) que, por exemplo, dados pretendidos já estão nos seus *buffers*. Neste contexto, podem em circunstâncias concretas serem geradas demasiadas interrupções, o que implicará demasiado tempo de processamento em rotinas de tratamento dessas mesmas interrupções. Por essa razão, algum *hardware*, como é o caso dos adaptadores *Gigabit Ethernet*, dispõem de mecanismos que permitem, em certa medida, controlar a geração de interrupções: os pacotes são reunidos num *buffer* durante um número predefinido de microssegundos, altura em que a interrupção é gerada. Tal constitui o *Interrupt Coalescing*.

Do dito anteriormente, deverá ser relativamente fácil inferir que ao aumentar o atraso até ser gerada a interrupção estaremos a diminuir o *overhead* inerente à transferência em causa; ao reduzi-lo estaremos a reduzir a latência, com a penalização de maior *overhead*. Obviamente, não podemos assumir linearidade em ambos os casos, pois no mundo do Sistema de Operação inúmeras variáveis terão de ser consideradas.

Posto isto, iremos agora explorar o conceito introduzido, recorrendo precisamente ao exemplo anterior[7]; para o conseguir, usaremos a ferramenta `ethtool` para ajustar o parâmetro `rx-usecs`. O valor por omissão, 20 microssegundos, é tido como o valor de compromisso entre latência e *overhead*. Será esse o valor a que nos referimos sempre que nada em contrário seja dito.

Os dois objectivos iniciais para esta primeira fase de testes da infra-estrutura de rede são determinar a taxa máxima de transferência e a latência minima para cada infra-estrutura e pilha protocolar: IB, IPoIB, IPoE e MXoE. As figuras 3.2 e 3.3 conferem-nos uma primeira amostra do efeito das opções em análise no desempenho.

[7]Que aliás é o único que estamos em condições de explorar.

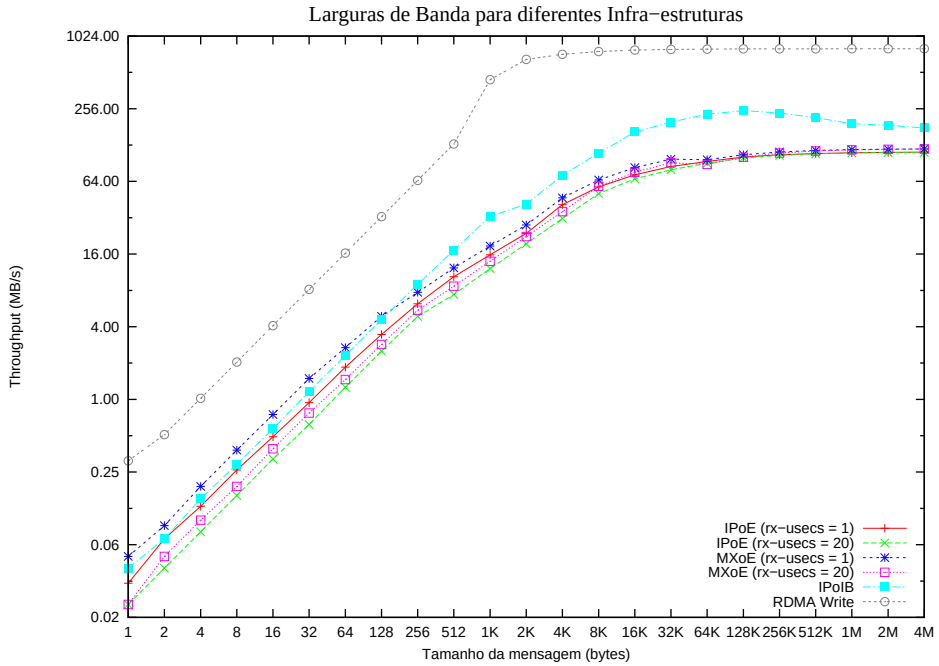


Figura 3.2: Larguras de Banda para as Infra-estruturas disponíveis no *cluster*. Recorremos às ferramentas `netperf` (IPoE, IPoIB), `omx_perf` (MXoE) e `ib_write_bw` (RDMA Write) de forma a obter os valores que nos permitiram construir o gráfico. Executámos as aplicações sem indicar quaisquer parâmetros, com excepção para a aplicação `netperf` que lançámos com os argumentos `-t TCP_RR --r $size,$size`, fazendo variar a variável `size` de 1 a 4194304 (de forma a obter valores para os diferentes tamanhos de mensagem). O mesmo se aplica para o gráfico da figura 3.3.

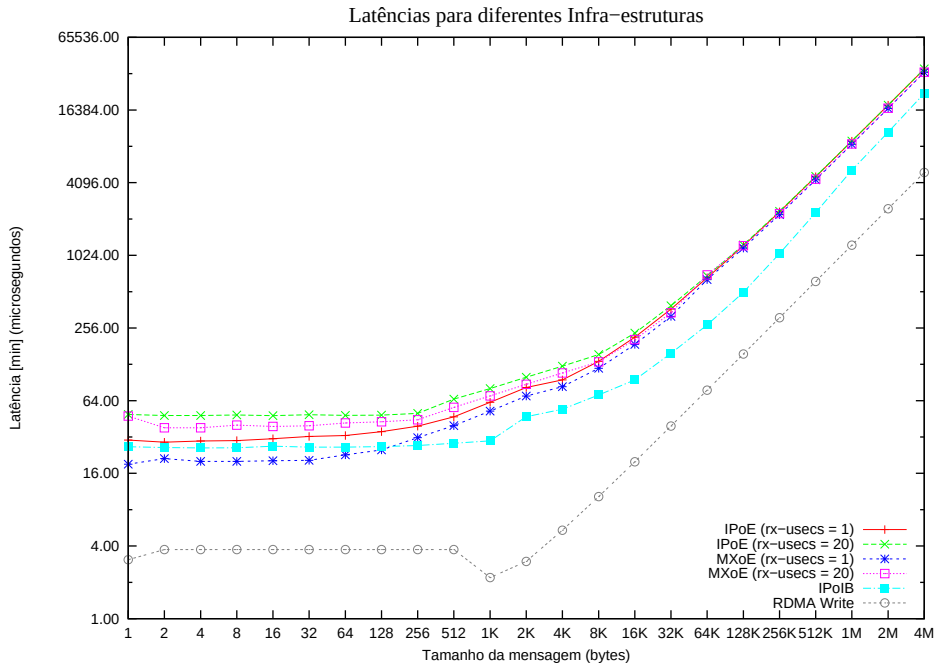


Figura 3.3: Latências para as Infra-estruturas disponíveis no *cluster*.

O gráfico da figura 3.3 evidencia a diferença, superior a uma ordem de grandeza, entre as latências exibidas pelas tecnologias *Gigabit Ethernet* e *InfiniBand*. Ainda, atendendo que a

curva de latência mais favorável para MXoE[8] se aproxima da curva para IPoIB, percebemos que o MXoE pode ser uma opção barata a considerar quando executamos aplicações sensíveis à latência mas não à largura de banda.

Concentremo-nos agora na aplicação `mpitests-IMB-MPI1`, e em particular no teste *Ping-Pong*; com ele podemos medir o tempo de transferência de um dado número bytes entre dois processos, possibilitando o cálculo da respectiva velocidade de transferência. Continuaremos a usar dois nós do cluster, excepto no caso onde forçamos o lançamento dos dois processos no mesmo CPU, com o objectivo de aferir o desempenho do MPI sobre memória partilhada. Note-se que sempre que usarmos uma das opções mencionadas, a não ser que algo em contrário seja dito, estaremos a forçar a utilização exclusiva dessa mesma opção.

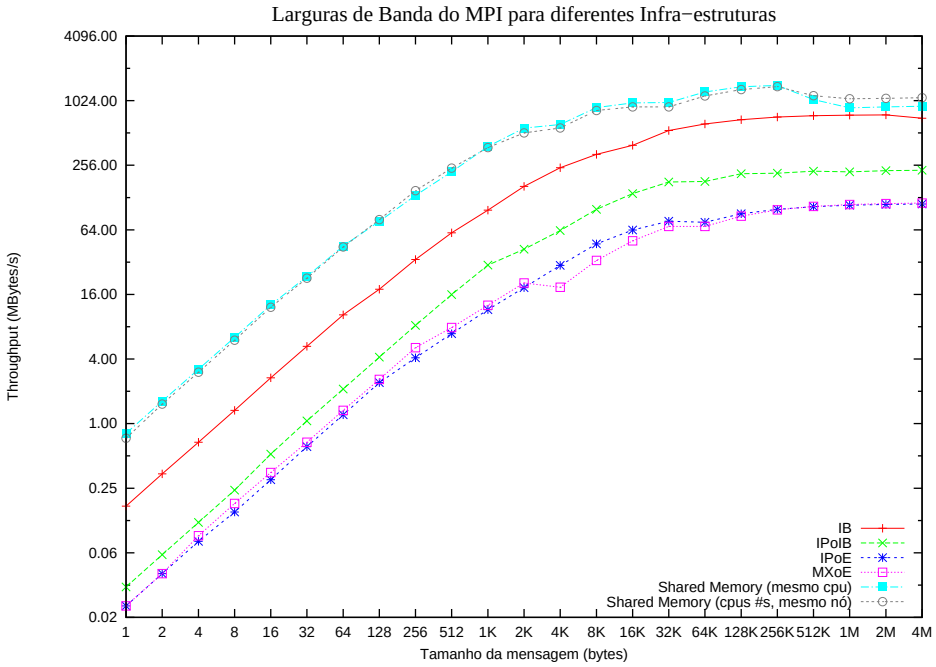


Figura 3.4: Teste de PingPong com a aplicação `mpitests-IMB-MPI1`, parte da *suite* Intel MPI Benchmarks.

A figura 3.4 permite-nos comparar o desempenho, medido em MBytes/s, das diferentes opções em função de diferentes trocas de dados, implicando diferentes tamanhos em bytes. De imediato é-nos possível, através de breve observação, retirar algumas conclusões:

- Uma diferença de performance entre o IB e o IPoE rondando os 640 MBytes/s;
- O débito tende a ser constante para mensagens de dimensão superior a 512 KB;
- Para SHM (*shared memory*) e mensagens excedendo os 256 KB – situação em que passa a ser benéfico correr os processos em CPUs diferentes – o débito cai para voltar a subir ligeiramente posteriormente. Acreditamos tratar-se de uma questão de caches;
- Se for necessária a utilização de TCP/IP, o IPoIB surge como uma excelente alternativa ao tradicional IPoE, superando-o em até 100 MBytes/s (aproximadamente). Essa vantagem é mais expressiva a partir do momento em que o *link* Ethernet começa a ficar saturado, o que visivelmente acontece por volta dos 64K;

[8] Com `rx-usecs` igual a 1 microssegundo.

- Para já não vislumbramos vantagem muito significativa em utilizar MXoE, ainda que este exceda em 2,31 MBytes/s o melhor resultado obtido com IPoE.

Tabela 3.3: Teste de PingPong com a aplicação `mpitests-IMB-MPI1`, em maior detalhe. Valores em MBytes/s (excepto coluna mais à esquerda).

bytes	IB	IPoIB	IPoE	MXoE	*Shared Memory*	
					mesmo cpu	cpus #s
1	0,17	0,03	0,02	0,02	0,81	0,73
2	0,34	0,06	0,04	0,04	1,62	1,52
4	0,67	0,12	0,08	0,09	3,21	3,01
8	1,33	0,24	0,15	0,18	6,37	5,99
16	2,67	0,52	0,30	0,35	12,75	12,16
32	5,25	1,06	0,61	0,67	23,50	22,66
64	10,32	2,10	1,21	1,33	45,18	44,28
128	17,86	4,17	2,41	2,58	76,73	79,73
256	33,85	8,23	4,11	5,11	133,88	148,15
512	60,10	15,96	6,90	7,87	221,90	240,59
1024	97,75	30,02	11,54	12,68	382,52	374,23
2048	162,64	42,18	18,51	20,45	568,36	512,96
4096	242,48	62,97	29,89	18,67	613,75	569,55
8192	323,34	99,32	47,29	33,15	882,32	829,97
16384	392,40	139,50	63,75	50,56	976,68	894,93
32768	539,37	178,78	77,06	68,90	984,35	896,42
65536	620,24	180,98	75,19	68,97	1233,63	1131,76
131072	680,93	213,30	90,43	85,97	1378,38	1299,80
262144	720,43	215,94	99,46	98,32	1419,27	1385,91
524288	740,45	225,31	105,17	105,71	1048,11	1139,97
1048576	750,64	222,32	108,57	110,12	878,71	1066,41
2097152	753,47	228,08	110,38	112,42	896,37	1077,25
4194304	702,25	230,01	111,29	113,60	908,24	1089,76

A tabela 3.4 mostra-nos as taxas máximas de transferência observadas para o teste de *PingPong* com a aplicação `mpitests-IMB-MPI1`, fazendo variar o número de microssegundos até ser gerada a interrupção após a recepção do primeiro pacote (`rx-usecs`).

Tabela 3.4: Débito no teste de *PingPong* em função do parâmetro `rx-usecs`.

	rx-usecs (μs)	Tempo (μs)	MBytes/s
MPI/IPoE	20	35934,25	111,31
	15	35933,95	111,32
	10	35920,25	111,36
	5	35898,15	111,43
	1	35890,39	111,45
MPI/MXoE	20	35209,00	113,61
	15	35174,25	113,72
	10	35154,09	113,78
	5	35116,15	113,91
	1	35074,56	114,04

Está bem patente que o MXoE é aquele que nos permite melhor usufruir da afinação do valor de `rx-usecs`, visto ter superado o débito inicial em 0,43 MBytes/s, mais que triplicando os 0,14 MBytes/s adicionais obtidos com IPoE. Se considerarmos que a opção *default* corresponde à utilização de IPoE com coalescência de 20 microssegundos, então verificamos ser possível, neste caso, um acréscimo de performance na ordem dos 2,4% ao adoptar MXoE e ajustando `rx-usecs` para 1 microssegundo.

Por último apresentamos o *benchmark* STREAM que nos permite conhecer ainda melhor o sistema em questão, ao avaliar o desempenho do par CPU/subsistema de memória.

Tabela 3.5: O *benchmark* `STREAM` permite-nos estimar a largura de banda efectiva da memória. Cada teste é executado 10 vezes, mas apenas o melhor resultado é usado. Resultados para um nó (4 *threads*).

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy:	5428,4218	0,0059	0,0059	0,0059
Scale:	5276,2689	0,0061	0,0061	0,0061
Add:	5927,2977	0,0081	0,0081	0,0082
Triad:	6043,1215	0,0080	0,0079	0,0080

3.2 SMP (cc-NUMA) – Sun Fire X4600 M2 Server

O servidor SMP que usaremos é um Sun Fire X4600 M2, construido há 3 ou 4 anos, que exibe grande parecença com o *cluster* em termos dos componentes individuais que utiliza. Concretamente, voltamos a dispor de 16 *cores* AMD Opteron, ainda que a sua frequência seja mais elevada em comparação com os *cores* dos servidores do *cluster*. Porém tal não será um obstáculo já que podemos facilmente ajustar a frequência dos primeiros de forma a igualar os 2.2GHz dos últimos, se assim o desejarmos.

Eis as suas características essenciais:

- 8× Dual-Core AMD Opteron 8220 (Santa Rosa), 2.8GHz, 90nm;
- 28 GB DDR2/667, ECC-registered;
- 2× Seagate ST914602SSUN146G 146GB, 10000 RPM, SAS;
- 2× Intel 82546EB Dual Port Gigabit Ethernet Controller.

Em relação às caches, as especificações são exactamente aquelas que apresentámos para os servidores do *cluster*, o que dispensa qualquer discussão adicional.

Antes de prosseguirmos com outros *benchmarks*, é importante fazer uma curta pausa para introduzir um factor cujo efeito pode afectar significativamente a performance de um sistema como o que aqui descrevemos, e que diz respeito à comunicação entre processadores. No caso do nosso sistema, essa comunicação faz-se através de ligações (*links*) *HyperTransport*, o *interconnect* utilizado pela AMD. Acontece que nem todos processadores estão directamente ligados entre si[9], fazendo com que seja necessário, por vezes, dar vários "saltos" (*hops*) entre processadores percorrendo uma dada distância (*hop count*). A figura 3.5 dá-nos a conhecer o número de *hops* entre os processadores (0 a 7).

[9] A justificação para tal sai fora do âmbito deste trabalho.

CPU	0	1	2	3	4	5	6	7
0	0	1	1	2	2	2	2	3
1	1	0	2	1	2	2	1	2
2	1	2	0	2	1	1	2	2
3	2	1	2	0	1	1	2	2
4	2	2	1	1	0	2	1	2
5	2	2	1	1	2	0	2	1
6	2	1	2	2	1	2	0	1
7	3	2	2	2	2	1	1	0

Figura 3.5: *Hops* entre processadores no servidor Sun Fire X4600 M2 (8-way).

Facilmente se conclui que não utilizando todos oito processadores, a escolha daqueles a usar não deverá ser indiferente, conforme ilustram os seguintes resultados referentes à execução do *benchmark* STREAM no CPU do nó indicado pela opção **--physcpubind** e utilizando memória local ao CPU do nó **--membind**.

 # hop count = 0, CPU0--CPU0

```
$ numactl --physcpubind=0 --membind=0 ./stream

Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:        1799.6719       0.0178       0.0178       0.0179
Scale:       1775.7896       0.0181       0.0180       0.0181
Add:         1775.3511       0.0271       0.0270       0.0271
Triad:       1815.7651       0.0265       0.0264       0.0266
```

 # hop count = 3, CPU0--CPU4

```
$ numactl --physcpubind=0 --membind=4 ./stream

Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:        1265.7154       0.0254       0.0253       0.0258
Scale:       1250.0953       0.0257       0.0256       0.0263
Add:         1281.1925       0.0376       0.0375       0.0382
Triad:       1318.4625       0.0365       0.0364       0.0367
```

Recuperaremos este tema na secção 5.13. Por último apresentamos o resultado obtido pelo mesmo *benchmark* desta feita utilizando os 16 núcleos disponíveis.

Tabela 3.6: *Benchmark* `STREAM` executado no servidor Sun Fire com diferentes números de *threads*. Observamos uma perda de desempenho ao aumentarmos o número de *threads* de 8 para 16.

	Rate (MB/s)			
Threads	Copy	Scale	Add	Triad
1	2961,8830	2827,3030	2694,8104	2955,2961
2	4706,5865	4497,6117	4581,4353	4713,8045
4	8288,1146	8036,0273	8267,3535	8251,4280
8	12539,0254	12450,6241	12483,8217	12473,7665
16	10741,7149	10858,1610	10755,2002	10781,6951

3.3 GPGPU – Nvidia Tesla C2050

A unidade Nvidia Tesla C2050 surgiu recentemente para dar continuidade à expansão do GPU enquanto plataforma de computação multi-propósito. Esta série promete a melhoria da performance em dupla precisão face às suas predecessoras, o que é relevante se considerarmos que o seu mercado natural se situa nos nichos HPC onde são executadas aplicações frequentemente contendo cálculos complexos e requerendo rigor.

O entusiasmo em torno destes *chips* advém de se verificar ser possível obter a mesma performance de processadores *quad-cores* recentes por um décimo do custo, e com um consumo energético vinte vezes inferior.
Apresentamos algumas das suas especificações:

- 448 CUDA *cores*, 1.15GHz, 515 Gflops (*double precision*);
- 3 GB GDDR5 (1.5GHz), 144 GB/s;
- 238W TDP (*Thermal design power*).

Capítulo 4

AMBER

O AMBER[1] é possivelmente um dos *packages* mais utilizados para a simulação de biomoléculas. Compreende um conjunto de aplicações para simulação em dinâmica molecular – particularmente indicadas para trabalhar com proteínas e ácidos nucleicos – que permitem modelar as interacções entre partículas através de um conjunto de parâmetros e funções analíticas – os campos de força.

Uma possível razão para utilizar o AMBER poderia ser o interesse em determinar a configuração mais estável de uma molécula[2], a qual se assume ser dada pela sua forma de menor energia. Tal envolve fazer variar, sucessivamente, uma série de parâmetros que caracterizam o sistema (e.g., ângulo entre dois átomos) e calcular para cada nova configuração a respectiva energia. A tarefa parece trivial, porém o número de átomos pode ascender às centenas de milhar.

Está portanto implícito um processo exaustivo, computacionalmente exigente e não determinístico. Felizmente as duas aplicações de simulação, PMEMD e SANDER, paralelizam a sua execução, recorrendo ao MPI e adoptando o modelo clássico *master/slave*.

Dado o potencial associado ao conhecimento que das simulações pode ser extraído, esforços têm sido feitos para desenvolver versões optimizadas destas aplicações; no nosso caso, iremonos concentrar em melhorar o desempenho da aplicação SANDER.

4.1 Caracterização

Nesta secção procuramos expandir o nosso conhecimento sobre as aplicações que o AMBER oferece, essencialmente através de uma análise quantitativa do seu código fonte. Concretamente, realizou-se uma contagem dos ficheiros correspondentes a código C/C++ (extensões .c e .cpp) e a código FORTRAN (extensões .f e .f90) e ainda do número de linhas de código presentes em cada linguagem. De mencionar que a contagem do número de linhas de código foi feita por via da aplicação `wc`[3] pelo que os números obtidos incluirão, certamente, linhas em branco e comentários. No entanto, e assumindo uma homogeneidade do código nestes aspectos, tais números serão representativos do número efectivo de linhas de código. Os valores apurados são apresentados nas figuras 4.1 e 4.2. Note-se que os valores apresentados para a coluna "AMBER11" são uma parte dos apresentados para "AMBER11 + AMBERTOOLS", da mesma forma que os valores para "SANDER" e "PMEMD" são uma parte daqueles apresentados para a coluna "AMBER11", e por conseguinte uma parte do total; as AMBERTOOLS incluem as bibliotecas BLAS e LAPACK.

[1] `http://ambermd.org/`

[2] A *suite* Amber suporta outros cálculos, que não serão descritos aqui visto exigirem conhecimentos consideráveis de Química para poderem ser apreciados. Como tal, optou-se por descrever a operação mais simples de ser compreendida pela pessoa sem formação específica na área, como é o caso do autor.

[3] Consultar `http://www.gnu.org/software/coreutils/manual` para mais informações.

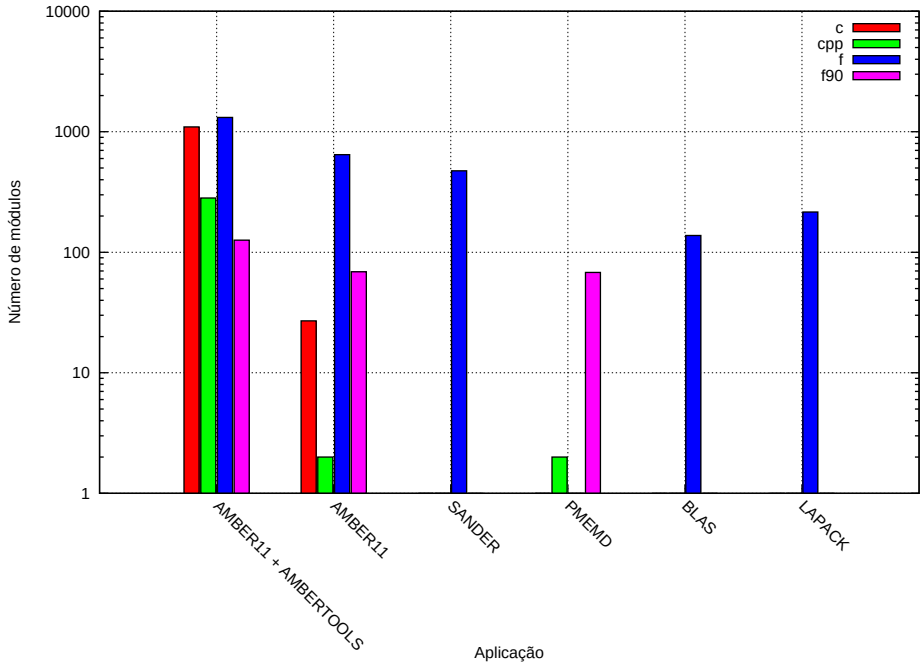


Figura 4.1: Número de módulos por cada linguagem usada na suite AMBER.

Da sua observação podemos retirar algumas conclusões, nomeadamente:

- No que diz respeito à totalidade do código, verifica-se que existe um equilíbrio marcado entre o código C/C++ e FORTRAN, uma vez que a soma das respectivas barras verde e vermelha é aproximadamente a mesma daquela das barras azul e lilás, para ambos os critérios considerados;

- A aplicação com que trabalhamos, o SANDER, é maioritariamente código FORTRAN (apenas uma fonte C);

- A grande maioria do código C++ diz respeito à aplicação PMEMD, ainda que as AMBERTOOLS tenham a si associado o maior número de fontes da linguagem em questão.

As tabelas seguintes apresentam os valores exactos utilizados para construir os gráficos previamente introduzidos.

Tabela 4.1: Número de módulos por cada linguagem usada na suite AMBER.

		c	cpp	f	f90
AMBER11 + AMBERTOOLS		1097	282	1314	126
AMBER11		27	2	646	69
AMBER11	SANDER	1	0	475	0
	PMEMD	1	2	0	68
AMBERTOOLS	BLAS	0	0	138	0
	LAPACK	0	0	216	0

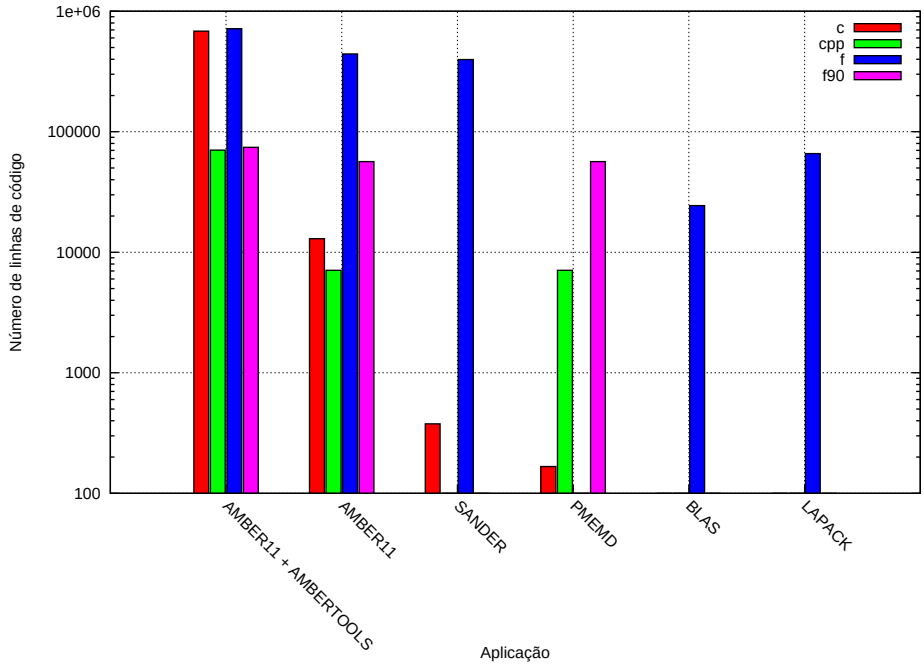


Figura 4.2: Número de linhas código em diferentes contextos da *suite* AMBER.

Tabela 4.2: Número de linhas código em diferentes contextos da *suite* AMBER.

		c	cpp	f	f90
AMBER11 + AMBERTOOLS		683500	70407	716978	74364
AMBER11		12963	7088	442477	56552
AMBER11	SANDER	377	0	397761	0
	PMEMD	167	7088	0	56530
AMBERTOOLS	BLAS	0	0	24388	0
	LAPACK	0	0	65847	0

4.2 Profiling MPI

Esta secção introduz a utilização das ferramentas do Oracle Solaris Studio (OSS), que usaremos de modo a obter um conjunto diversificado de informações respeitantes aos eventos ocorridos durante a execução (traço) de simulações realizadas com o PMEMD e SANDER. Para tal, iremos recorrer às aplicações `collect` e `analyzer` (OSS Performance Analyzer), que nos possibilitam, respectivamente, obter e visualizar tais informações. Estamos particularmente interessados em explorar a capacidade que a ferramenta `collect` possui para recolher uma série de indicadores relativos à comunicação entre processos MPI — funcionalidade implementada com o *package* VampirTrace 5.5.3 (opensource). Para o fazer, seleccionámos um exemplo a partir de uma colectânea de *benchmarks* disponíveis no site oficial do AMBER[4], correspondendo a uma simulação com a proteína da mioglobina — associada à fixação de oxigénio no sangue. Optámos por uma simulação algo complexa, temendo obter resultados não confiáveis, ou pouco representativos, se tivéssemos optado por um exemplo minimalista.

Alertamos o leitor que todos os testes que consideramos nas subsecções seguintes foram

[4] `http://ambermd.org/amber11_bench_files/Amber11_Benchmark_Suite.tar.gz`

executados sobre máquinas virtuais que oportunamente instalámos e configurámos em cada um dos nós do *cluster*. Tal não deverá alterar, no essencial, as conclusões que retiramos. Acontece que tais testes foram executados numa fase do trabalho onde ainda preparávamos a migração para as máquinas reais; planeámo-lo dessa forma atendendo à necessidade de acautelar o bom funcionamento e o estado[5] dessas máquinas. Adicionalmente salvaguardámos a hipótese de, no futuro, realizar outros testes, se desejável, possibilitando a comparação com os resultados a obter nas máquinas reais.

4.2.1 Teste numa VM com `pmemd.MPI` (sobre *shared memory*)

Começamos por executar o exemplo num único nó, com dois processos (NP = 2), utilizando a aplicação `pmemd.MPI`. Em todas as simulações será usada uma imagem do AMBER compilada com o gcc-4.4.5 e ligada com as ferramentas hpc-8.2.1c-gnu, uma implementação MPI da Oracle[6].

Para obter o traço devemos executar o comando que indicamos, na directoria do exemplo:

```
collect -M OMPT -p low mpirun -np 2 -- ...
```

Posteriormente lançamos a aplicação `analyzer` e puderemos visualizar os ecrãs que exibimos, e discutimos, de seguida.

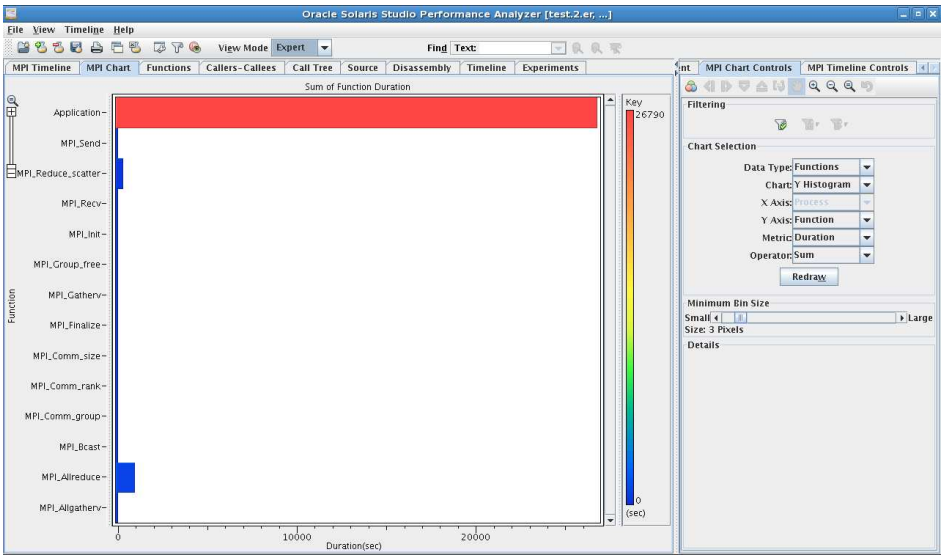


Figura 4.3: MPI Chart – OSS Performance Analyzer.

4.2.1.1 MPI Chart

Este separador mostra-nos um gráfico que nos permite avaliar qual o peso das operações MPI em relação às demais, considerando o tempo total de execução. Para a simulação em causa, executada com dois processos MPI e tendo demorado aproximadamente 4 horas, a figura 4.3 revela a primazia indiscutível do tempo de computação sobre o tempo de comunicação. Ficamos ainda a saber que a operação MPI que mais tempo consome no total é a MPI_Allreduce, a qual combina valores de todos os processos e distribui o resultado a todos processos.

[5] Referimo-nos à "limpeza" do sistema, que em grande medida implica não instalar mais pacotes que o necessário e saber exactamente quais as configurações a realizar.

[6] Para maior compatibilidade com a aplicação de profiling.

4.2.1.2 Function

Este separador apresenta-nos os tempos totais para cada uma das funções chamadas pela aplicação. Uma observação apressada da figura 4.4 levar-nos-ia a concluir que as operações MPI representam aproximadamente apenas 5% do tempo total apresentado[7] – confirmando a observação anterior – e que são transferidos 104 bytes. Todavia, a informação apresentada pelo OSS Performance Analyzer relativamente ao número de bytes transferidos, bem como ao número de mensagens enviadas e recebidas, diz unicamente respeito às comunicações ponto-a-ponto[8], pelo que a conclusão apressada rapidamente terá de ser revista.

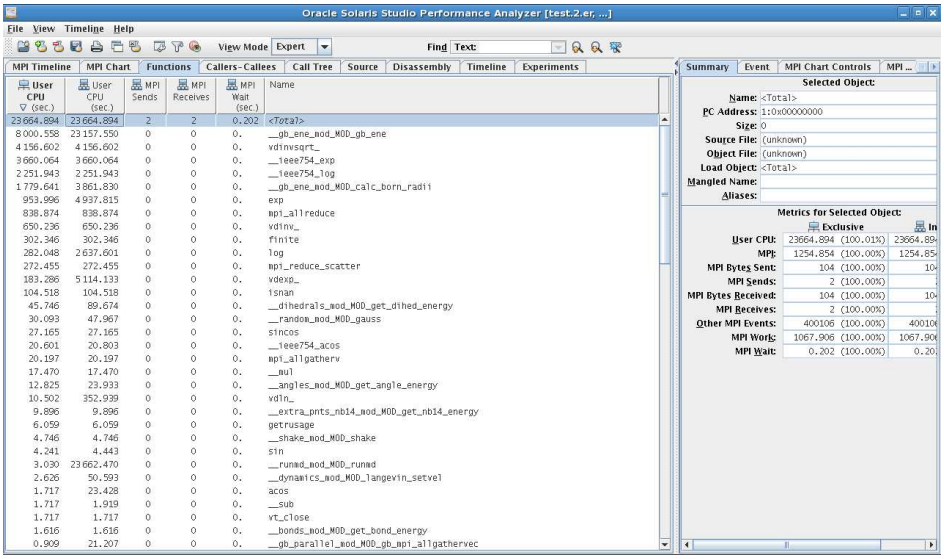


Figura 4.4: Functions – OSS Performance Analyzer.

4.2.1.3 MPI Timeline

Este separador permite-nos ter uma visão geral das chamadas MPI de uma aplicação ao longo da sua execução. A figura 4.5 exibe os resultados recolhidos para a simulação em análise: as operações MPI encontram-se representadas em tons de azul, e o restante código corresponde às áreas a cinzento.

As operações MPI são visivelmente pouco significativas em tempo, o que poderá eventualmente reflectir o cuidado em minimizar o tempo de comunicação (e sincronização) entre processos. Não devemos contudo ignorar que executámos esta simulação sobre memória partilhada, pois ambos os processos foram lançados na mesma máquina. Tal reduzirá substancialmente, admitimos, o tempo de comunicação.

4.2.2 Teste em duas VMs (uma por nó) com `pmemd.MPI`

Para observar o efeito da troca de mensagens sobre uma rede (IPoE) a simulação foi repetida, desta vez usando 2 nós, cada um hospedando uma VM, e em cada VM foram lançados 2 processos `pmemd.MPI`. Neste teste, a comunicação fez-se exclusivamente por rede, i.e., aos processos residentes no mesmo nó foi negada a possibilidade de comunicar por memória partilhada.

[7] Ver lado direito.

[8] Na ajuda da aplicação, em "MPI Tracing Metrics", podemos ler: "MPI byte and message counts are currently collected only for point-to-point messages; they are not recorded for collective communication functions.".

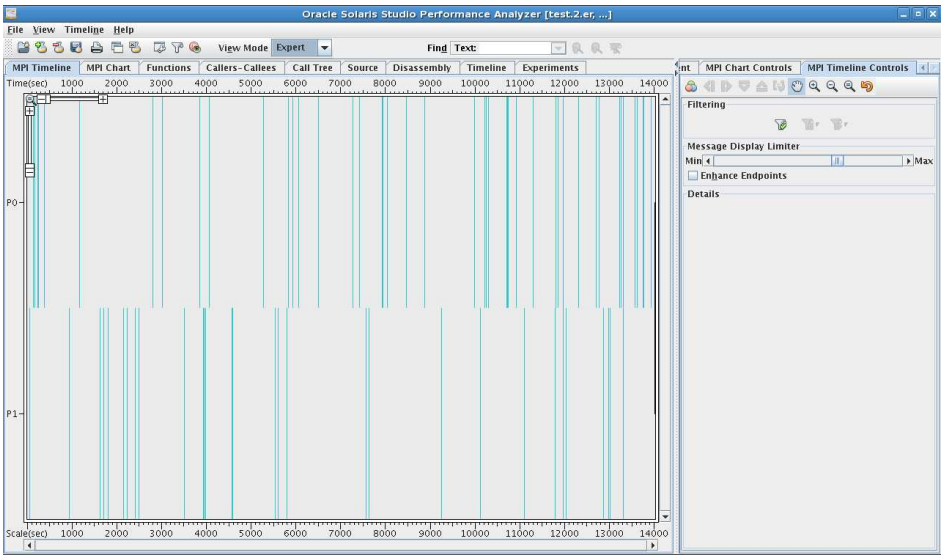


Figura 4.5: MPI Timeline – OSS Performance Analyzer.

4.2.2.1 MPI Timeline

O *timeline* para este teste mostra um cenário bastante diferente daquele que foi apresentado para a execução em memória partilhada: a mera observação da figura 4.6 permite constatar imediatamente o grande aumento no número de operações MPI efectuadas.

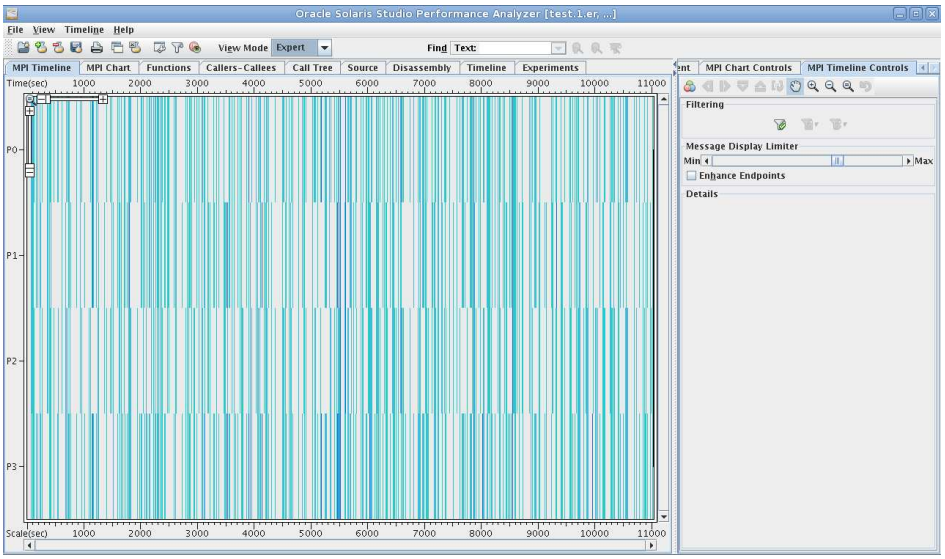


Figura 4.6: MPI Timeline – OSS Performance Analyzer.

4.2.2.2 MPI Chart & Function

As figuras 4.7 e 4.8 permitem imediatamente concluir que a aplicação pmemd.MPI usa essencialmente primitivas de comunicação colectivas com agregação de valores, sendo que (fig. 4.8) o número de operações individuais *send*/*receive* é apenas 6.

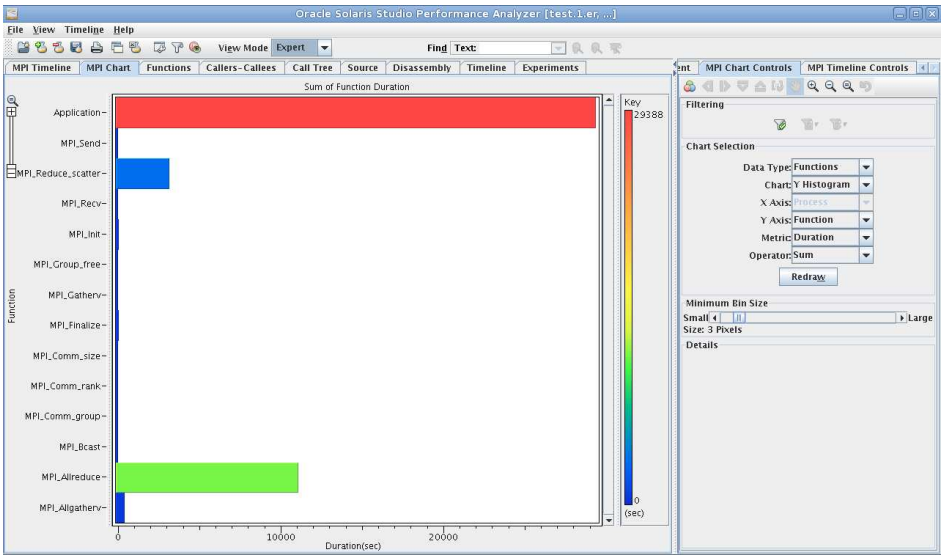


Figura 4.7: MPI Chart – OSS Performance Analyzer.

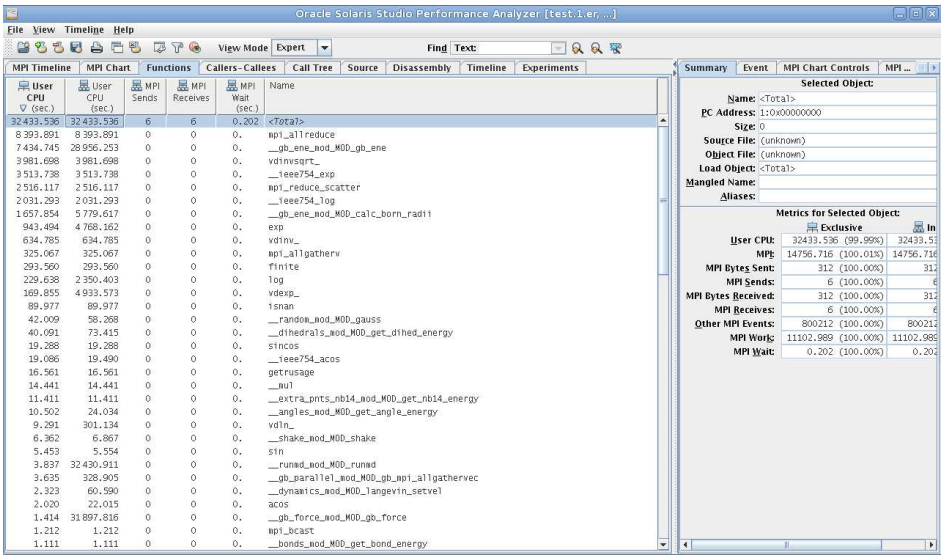


Figura 4.8: Functions – OSS Performance Analyzer.

4.2.2.3 Padrão de Comunicação na aplicação PMEMD

A análise efectuada demonstra que a rotina MPI mais utilizada corresponde à operação de redução conjunta (*Allreduce*), a qual combina valores de todos os processos envolvidos. Sendo assim, concluímos que não existirá qualquer optimização da colocação dos processos em nós para favorecer o processo de comunicação a ser explorado (a não ser que outras simulações provem o contrário).

4.2.2.4 SANDER

O grupo de testes anteriores foi agora repetido, nas mesmas condições, com o `sander.MPI` para que fosse possível observar eventuais diferenças em relação ao *pmemd.MPI*.

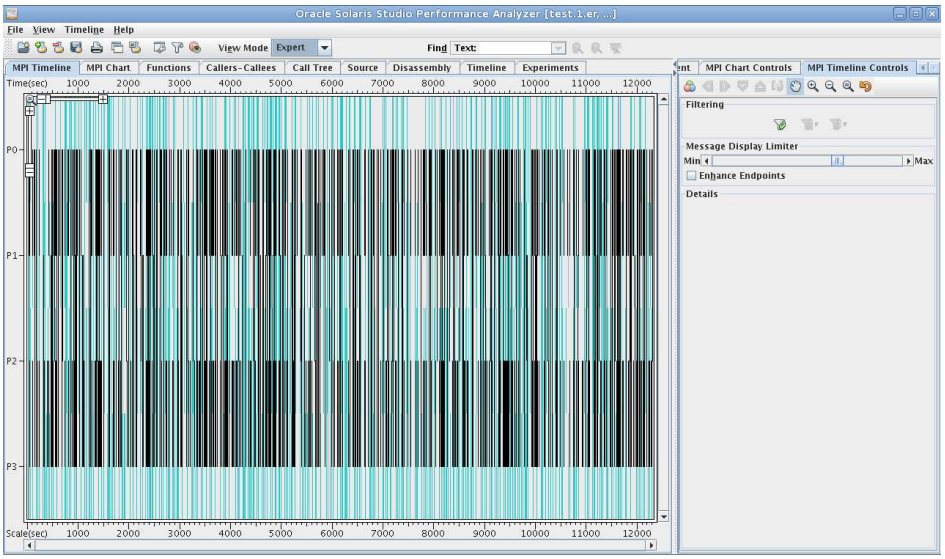


Figura 4.9: MPI Timeline – OSS Performance Analyzer.

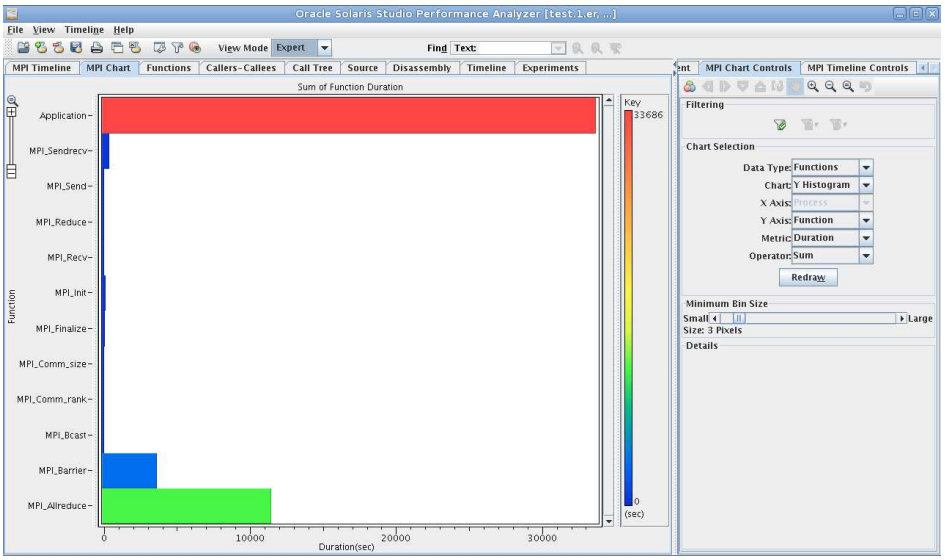


Figura 4.10: MPI Chart – OSS Performance Analyzer.

O resultado, capturado pelas figuras 4.9 a 4.11, mostra uma nova realidade não pelo tempo cumulativo das operações MPI, que aliás se mantém em aproximadamente 50% do total, mas sim em termos da utilização dos canais de comunicação, a qual cresce substancialmente, atingindo os 8.35 GB de dados. Se até então o número de MPI *sends*/*receives* era desprezável, verifica-se que excede agora os 400 mil para a simulação em causa. Observando que o número de outros eventos MPI é aproximadamente o mesmo que para o caso anterior com PMEMD, parece inevitável que o SANDER seja consideravelmente mais sensível às características da topologia de interconexão entre os nós.

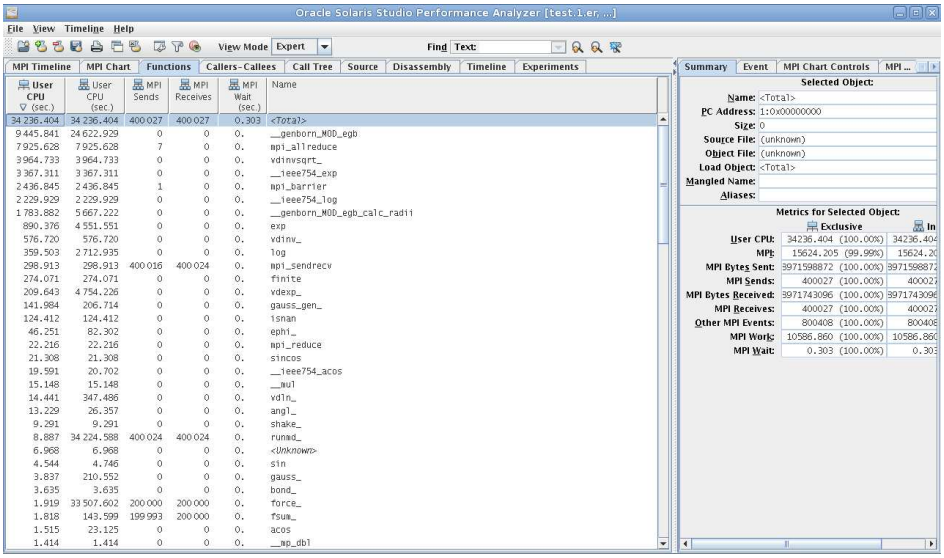


Figura 4.11: Functions – OSS Performance Analyzer.

Capítulo 5

Estudos de Optimização

Neste capítulo serão apresentados os estudos que exploram os percursos de optimização previamente discutidos, para determinar eventuais oportunidades de incrementar a performance em relação à configuração base do AMBER.

5.1 Execução Paralela

Dada a sua natureza computacionalmente exigente, ao executarmos as aplicações PMEMD e SANDER estamos fundamentalmente interessados na execução paralela, o que, no caso destas duas aplicações, se traduz na utilização das versões MPI ou CUDA[1].

O estudo da versão MPI será o mais abrangente dos que irão ser apresentados ao longo desta exposição, confrontados que fomos com a impossibilidade de realizar amplos estudos em *single-node*, dado o tempo que tal exigiria, para um exemplo suficientemente realista.

Para o efeito, recorremos a uma simulação disponível na página do AMBER [2], denominada JAC Production NVE.

Como preparação, foi necessário criar um *bash script* que simultaneamente compilasse e testasse a performance de um conjunto de imagens do Amber, utilizando em cada *round* um conjunto diferente de *flags* de optimização (e eventualmente outros parâmetros pertinentes) a definir pelo utilizador.

5.2 Optimizações do GCC

A utilização de vários núcleos *Opteron* revelou-se em relação às optimizações do compilador indispensável para a realização de inúmeras simulações que expõem a nossa abordagem, cuja essência é capturada pelas seguintes questões: qual a influência individual de cada optimização que compõe um nível de optimização do GCC? Existirão optimizações que prejudicam o resultado final, não obstante o resultado global ser positivo?

Ao dar resposta a estas questões, o nosso objectivo é excluir eventuais optimizações que desfavoreçam o resultado. Tal pode parecer um paradoxo, mas é na nossa perspectiva possível[3]. Muitas vezes a questão prende-se com o facto de o compilador precisar de estabelecer determinadas condições, como o alinhamento dos dados ao utilizar instruções SSE, que requerem a execução de instruções suplementares. Por essa e por outras razões o saldo final

[1] No início deste trabalho estavamos convencidos que uma versão Open MP estava também disponível; contudo viemos a verificar que apenas uma ferramenta da *suite* AMBERTOOLS tem suporte para Open MP.

[2] http://ambermd.org/amber10.bench1.html#jac, ponto 2.

[3] É de facto possível. Ao consultar o manual do GCC 4.4.6, secção 3.10 – *Options That Control Optimization* – podemos encontrar referências a esse facto. A título de exemplo, sobre a opção *-falign-labels*, activada nos níveis -O2 e -O3, o manual menciona: "*This option can easily make code slower, because it must insert dummy operations for...*".

Tabela 5.1: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.4.5, de acordo com a opção indicada.

Opção		Tempo (s)	*Speedup* (×)
-O0		1086,866	---
-O1		579,583	1,875
-O2		497,054	2,187
-O3		504,177	2,156
-O1	-fauto-inc-dec	1085,022	1,002
	-fcprop-registers	1084,84	1,002
	-fdce	1091,798	0,995
	-fdefer-pop	1087,785	0,999
	-fdelayed-branch	1099,269	0,989
	-fdse	1079,934	1,006
	-fguess-branch-probability	1086,275	1,001
	-fif-conversion2	1089,411	0,998
	-fif-conversion	1088,661	0,998
	-finline-small-functions	1089,121	0,998
	-fipa-pure-const	1080,801	1,006
	-fipa-reference	1089,14	0,998
	-fmerge-constants	1091,988	0,995
	-fsplit-wide-types	1085,85	1,001
	-ftree-builtin-call-dce	1090,205	0,997
	-ftree-ccp	1082,775	1,004
	-ftree-ch	1094,496	0,993
	-ftree-copyrename	1082,962	1,004
	-ftree-dce	1090,713	0,996
	-ftree-dominator-opts	1084,08	1,003
	-ftree-dse	1096,122	0,992
	-ftree-fre	1100,73	0,987
	-ftree-sra	1096,006	0,992
	-ftree-ter	1109,681	0,979
	-funit-at-a-time	1095,969	0,992
	-fomit-frame-pointer	1091,853	0,995

pode ser negativo. Contudo, a nossa avaliação apenas pode ser feita ao nível do individual, significando que não dispomos de maneira de testar todas as combinações possíveis, dado o número de optimizações existentes e a complexidade da nossa aplicação. Ainda assim, através de um método *naïve*, partindo dos resultados individuais, provaremos ser possível determinar uma combinação que supera o desempenho da compilação *default*.

A tabela 5.1 apresenta os tempos de execução para a versão MPI do SANDER, compilada com o GCC 4.4.5 para um conjunto de *flags* de optimização[4] e executada utilizado o Open MPI 1.4.3.

[4]O resultado para as flags -fschedule-insns -fschedule-insns2 não está disponível devido à obtenção de um erro de compilação a elas associado: _ncsu-rmsd.f:169: error: unable to find a register to spill in class 'CREG'.

Opção		Tempo (s)	*Speedup* (×)
-O2	-fthread-jumps	1088,867	0,998
	-falign-functions -falign-jumps	1083,074	1,004
	-falign-loops -falign-labels	1095,218	0,992
	-fcaller-saves	1082,685	1,004
	-fcrossjumping	1089,728	0,997
	-fcse-follow-jumps -fcse-skip-blocks	1092,852	0,995
	-fdelete-null-pointer-checks	1088,245	0,999
	-fexpensive-optimizations	1091,272	0,996
	-fgcse -fgcse-lm	1084,815	1,002
	-findirect-inlining	1092,787	0,995
	-foptimize-sibling-calls	1091,133	0,996
	-fpeephole2	1094,402	0,993
	-fregmove	1095,658	0,992
	-freorder-blocks -freorder-functions	1095,891	0,992
	-frerun-cse-after-loop	1096,41	0,991
	-fsched-interblock -fsched-spec	1083,017	1,004
	-fschedule-insns -fschedule-insns2	---	--
	-fstrict-aliasing -fstrict-overflow	1090,549	0,997
	-ftree-switch-conversion	1099,695	0,988
	-ftree-pre	1091,234	0,996
	-ftree-vrp	1088,383	0,999
-O3	-finline-functions	1095,831	0,992
	-funswitch-loops	1085,751	1,001
	-fpredictive-commoning	1093,394	0,994
	-fgcse-after-reload	1096,991	0,991
	-ftree-vectorize	1084,742	1,002
	-fipa-cp-clone	1082,865	1,004

Da sua análise, podemos retirar duas conclusões: em primeiro, e ao contrário do que poderíamos supor, não existem duas optimizações distintas tais que a diferença entre os respectivos tempos de execução seja significativa; em segundo, não existe nenhuma optimização que supere, mais do que residualmente, o resultado obtido quando se compilou sem quaisquer optimizações (-O0). Assim sendo, somos forçados a concluir que a melhoria que efectivamente se obtém ao utilizar um dos níveis -O1, -O2 ou -O3 advém exclusivamente da acção conjunta de uma série de optimizações individuais. Os resultados obtidos parecem também confirmar a ideia, algo generalizada, que o nível -O2 representa na maioria dos casos o melhor compromisso; no nosso caso, obtivemos um pior resultado ainda que por curta margem[5] ao utilizar -O3 em vez de -O2.

A tabela 5.2 permite-nos prosseguir a discussão, exibindo os resultados relativos ao desempenho do `sander.MPI` quando compilado de forma a utilizar instruções optimizadas para o CPU local. O intuito de tais testes foi confirmar que não existem diferenças expressivas entre o uso das opções `opteron` e `native`. Não obstante os resultados demonstrarem, para todos os efeitos, que se relevou indiferente qual a *flag* utilizada de entre as quatro listadas, os testes subsequentes utilizam a primeira opção visto ser mais explícita.

O ensaio não poderia estar completo sem alguma experimentação preliminar com as instruções SSE, verificando a diferença entre utilizar apenas instruções aritméticas do *set* SSE ou, se possível, gerar em simultâneo instruções SSE e para o co-processador 387. Todavia,

[5]Esta não foi a única situação. Ao observar a tabela 5.4 percebemos o nível -O3 não consegue ultrapassar o -O2 mesmo com o beneficio da *flag* -mtune=opteron. Tal apenas sucede na última configuração, que será discutida mais adiante.

Tabela 5.2: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.4.5, de acordo com a opção indicada — CPU tuning.

Opção	Tempo (s)	*Speedup* (×)
-mtune=opteron	1077,3	1,009
-march=opteron	1078,535	1,008
-mtune=native	1087,822	0,999
-march=native	1076,401	1,010

Tabela 5.3: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.4.5, de acordo com a opção indicada — Uso de instruções SSE.

Opção	Tempo (s)	*Speedup* (×)
-mfpmath=sse -mno-sse -msse2 -mmmx	1088,178	0,999
-mfpmath=sse,387 -mno-sse -msse2 -mmmx	1090,282	0,997

os resultados evidenciados pela tabela 5.3 não permitem vislumbrar, de momento, nenhum potencial de optimização. Oportunamente regressaremos a este tópico.

Confrontados com estes resultados, a tarefa de determinar uma melhor configuração para a compilação não se encontra facilitada. De certa forma, o nosso problema assemelha-se a uma procura cega, já que não dispomos de melhor informação do que a aqui apresentada. Neste contexto, deliberou-se que a hipótese seria tentar excluir as optimizações que, a nível individual, obtiveram o pior resultado, procurando assim atingir um resultado global melhor. Obviamente, devemos ter em consideração que a eliminação das n piores optimizações não nos conduzirá necessariamente ao melhor resultado. Porém, não antevendo uma heurística mais inteligente e muito limitados em poder computacional que estamos, é a solução possível.

Felizmente, a tabela 5.4 mostra-nos que não tomámos uma má opção. De facto, conseguimos obter um resultado próximo dos 5% — que se traduz em melhor desempenho em relação à compilação *default* — ao excluir as 5 piores optimizações e simultaneamente impondo o *tuning* do código para o processador instalado.

Não obstante podermos obter resultados significativos sempre que utilizamos as optimizações do compilador no caso de aplicações paralelas, podemos questionar-nos qual a influência dos algoritmos paralelos têm sobre a capacidade das optimizações melhorarem o desempenho. A tabela 5.5 apresenta alguns resultados relativos a execução em sequencial e demonstra que aplicação sai beneficiada na mesma medida que anteriormente, já que os *spe-*

Tabela 5.4: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.4.5, de acordo com a opção indicada — Determinando uma configuração paralela melhor.

Opção	Tempo (s)	*Speedup* (×)
-O3 -mtune=generic (Amber11 *default*)	507,514	---
-O3 -mtune=opteron	499,376	1,016
-mtune=opteron -O3 -fno-tree-ter -fno-tree-fre -fno-tree-switch-conversion -fno-delayed-branch -fno-gcse-after-reload	485,965	1,044

Tabela 5.5: Tempos de execução com a aplicação `sander`. AMBER compilado com gcc-4.4.5, de acordo com a opção indicada — Execução sequencial.

Opção	Tempo (s)	*Speedup* (×)
-O3 -mtune=generic (Amber11 *default*)	4449,145	---
-O3 -mtune=opteron	4382,294	1,015
-mtune=opteron -O3 -fno-tree-ter -fno-tree-fre -fno-tree-switch-conversion -fno-delayed-branch -fno-gcse-after-reload	4273,333	1,041

edups obtidos em relação à compilação *default* são semelhantes àqueles obtidos na execução paralela. Concluímos que o paralelismo não influencia as optimizações neste caso. Tal valida o estudo mais exaustivo apresentado inicialmente.

5.3 GCC 4.1.2

Até agora temos estado a utilizar a versão 4.4.5 do GCC, por nós compilada aquando da preparação do ambiente de trabalho, e continuaremos a fazê-lo. À data, assumimos que esta versão dificilmente seria pior que a sua predecessora em relação à performance do código produzido – de facto, espera-se que a qualidade do software aumente, ou no mínimo se mantenha igual entre as *releases*; porém nem sempre é o caso. Ainda assim, a versão que vem de origem com o CentOS 5.5 não foi removida do sistema, tendo em vista um ensaio comparativo entre as duas.

Nesta secção efectuamos esse estudo, motivados pelo facto de saber que o utilizador comum certamente recorrerá versão à distribuída com o CentOS (neste caso 4.1.2). Procuramos, portanto, saber se existe algum benefício ao utilizar a versão 4.4.5 para além da eventual introdução de novas funcionalidades ou ainda a correcção de *bugs* que tenham entretanto sido detectados. Para o efeito, voltámos a utilizar a aplicação SANDER, agora compilada com o gcc 4.1.2, e repetimos os testes apresentados pelas tabelas 5.4 e 5.5, nos mesmos moldes[6]. As tabelas 5.6 e 5.7 demonstram a superioridade da versão que temos vindo a usar.

Tabela 5.6: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.1.2, de acordo com a opção indicada.

Opção	Tempo (s)		*Speedup* (×)
	gcc-4.1.2	gcc-4.4.5	
-O3 -mtune=generic	558,689	507,514	1,101
-O3 -mtune=opteron	555,797	499,376	1,113

Os resultados são tão expressivos que dispensam qualquer argumentação adicional. É um dado incontornável que o gcc-4.4.5 produz código melhor em relação ao seu antecessor. Contudo, não dispensamos conhecer quais as razões, ou circunstâncias, que levam ao resultado agora apresentado. Como tal, recorremos a uma ferramenta de profiling disponibilizada pelo fabricante AMD, denominada *AMD CodeAnalyst*, que nos permite utilizar um conjunto de "contadores", realizados ao nível *hardware*, de forma a contabilizar eventos como cache *misses*, etc.

[6] O terceiro teste não pôde ser repetido uma vez que o gcc 4.1.2 não suporta todas a opções indicadas, concretamente a optimização -ftree-switch-conversion. Para maior detalhe, consultar o manual [17].

Tabela 5.7: Tempos de execução com a aplicação `sander`. AMBER compilado com gcc-4.1.2, de acordo com a opção indicada — Execução sequencial.

Opção	Tempo (s)		*Speedup* (×)
	gcc-4.1.2	gcc-4.4.5	
-O3 -mtune=generic	4972,408	4449,145	1,118
-O3 -mtune=opteron	4858,394	4382,294	1,109

Como tivemos oportunidade de referir anteriormente, a performance de uma aplicação está directamente associada com uma boa utilização da hierarquia de memória e muito em particular das caches; porque o preço a pagar, em ciclos de relógio, é bastante alto sempre que deixamos a zona de conforto das caches – já que as memórias que lhes sucedem na hierarquia são bastante mais lentas – deve ser possível observar algumas diferenças nos acessos às caches entre as execuções com gcc-4.1.2 e gcc-4.4.5, possivelmente com a mesma expressão que os resultados anteriores. A tabela 5.8 apresenta os resultados obtidos com a utilização do CodeAnalyst; nesta, os 4 agrupamentos correspondem a 4 tipos de testes realizados, respectivamente medindo a performance, os acessos à cache L2, os acessos às *data caches* e os acessos às *instruction caches*. Cada um destes tipos de testes foi executado em momentos diferentes e os seus resultados representam a informação recolhida durante toda a execução da respectiva aplicação[7]. Note-se que não terá necessariamente de existir uma correspondência entre valores relacionados em diferentes grupos, já que dentro de cada grupo os testes são realizados com parâmetros específicos.

Prosseguindo, da análise da tabela mencionada podemos retirar algumas conclusões:

- Observando o número de "*Retired Instructions*", concluimos que a versão compilada com o gcc-4.4.5 executa, em média, menos 9,64% instruções realmente necessárias[8] que a versão compilada com o gcc-4.1.2; podemos então afirmar que este último gera código com mais instruções do que o que seria necessário para obter o mesmo resultado;

- Observa-se que a versão compilada com o gcc-4.4.5 faz com que o processador esteja parado em menos ciclos de relógio (*CPU clocks not halted*), concretamente 10,38% menos, o que sugere melhor utilização dos recursos e poderá significar maior disponibilidade de dados na proximidade do processador;

- No geral os resultados parecem indicar que a versão compilada com gcc-4.4.5 sai triunfante nos acessos às caches. Destacamos:

  – Redução substancial dos acessos à *data cache* (-23,73%);

  – Diminuição apreciável do número de instruções obtidas da *instruction cache* (-11,13%), acompanhada pela supressão pujante (-39,55%) dos *misses* nessa mesma cache e pela subida em 12,74% do número de *hits* na L2 ITLB após *miss* na ITLB de nível anterior.

Por essa razão, deixa-se à consideração do leitor a recomendação de compilar o AMBER com uma versão relativamente recente do GCC.

[7] As tabelas subsequentes, apresentadas ao longo deste trabalho, seguirão a mesma lógica.

[8] Os processadores modernos implementam uma técnica conhecida por execução especulativa, que procura reutilizar os recursos executando instruções antecipadamente que, não obstante poderem incrementar a performance, podem-se provar desnecessárias do ponto de vista da aplicação em execução.

Tabela 5.8: Profiling com AMD CodeAnalyst para a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com "-O3 opteron".

Evento	**gcc-4.1.2**	**gcc-4.4.5**	
CPU clocks not halted	15012602	13454326	-10,38%
Retired branch instructions	9999660	9621098	-3,79%
Retired mispredicted branch inst.	429938	437184	+1,69%
L2 fill/writeback	2061746	1959887	-4,94%
L2 cache misses	201091	208317	+3,59%
Requests to L2 cache	1889678	1819512	-3,71%
Retired instructions	13286470	11871745	-10,65%
Data cache accesses	6902282	5264424	-23,73%
Data cache misses	1571940	1521172	-3,23%
Data cache refills from L2 or system	1570682	1517012	-3,42%
L1 DTLB miss and L2 DTLB miss	23196	24524	+5,73%
L1 DTLB miss and L2 DTLB hit	1557930	1605746	+3,07%
Misaligned accesses	92388	92294	-0,10%
Retired instructions	13293702	12023452	-9,56%
Instruction cache fetches	5522510	4907702	-11,13%
Instruction cache misses	38040	22996	-39,55%
L1 ITLB miss and L2 ITLB hit	5228	5894	+12,74%
L1 ITLB miss and L2 ITLB miss	416	104	-75,00%
Retired instructions	13342072	12179612	-8,71%

5.4 Compiladores Intel

Aqui realizamos a compilação do AMBER com as ferramentas da Intel, tendo por objectivo a comparação com o gcc-4.4.5 — em termos da performance da aplicação que temos vindo a usar. Estaremos particularmente interessados em averiguar a capacidade de vectorização do compilador da Intel, mas tal discussão será remetida para a secção 5.7.

Tabela 5.9: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado de acordo com a opção indicada. *Speedup* ao usar Intel.

Opção	Intel (s)	gcc-4.4.5 (s)	*Speedup* (×)
-O0	1388,253	1086,866	0,783
-O1	508,349	579,583	1,140
-O2	480,498	497,054	1,034
-O3	488,769	504,177	1,032
Amber 11 *default*	480,838	507,514	1,055

A tabela 5.9 apresenta os tempos de execução relativos a algumas opções de compilação oferecidas pela *suite* Intel, e a sua comparação com os resultados homólogos obtidos com o gcc-4.4.5. Perante estes resultados, não podemos estar para já muito entusiasmados com a utilização das ferramentas de compilação Intel. De facto, apenas superou de forma apreciável

o gcc-4.4.5 no nível de optimização -O1; de resto, mostra-se incapaz de fazer melhor do que uma média de 4% nas configurações testadas.

Note-se que a compilação *default*, quando são utilizados os compiladores Intel, determina a activação das flags -ip -O3 -xHost. Neste conjunto, a primeira flag indica a realização de uma técnica de optimização que considera todas as dependências de um módulo fonte, em vez de olhar apenas para cada entidade (função, sub-rotina) isoladamente, e tem por nome *Interprocedural Optimization* (-ip). Note-se que a compilação *default* com GCC não inclui semelhante estratégia de optimização. Por essa razão, o resultado favorável obtido pelos compiladores da Intel na configuração *default*, na ordem dos 6%, sai enfraquecido uma vez que a comparação com o GCC não é, nesse caso, inteiramente justa.

Ainda em relação à experimentação com as ferramentas Intel, procurámos alcançar uma melhor performance aplicando a técnica anterior na vertente em que todas as partes da aplicação são tratadas como uma só (-ipo), do ponto de vista da optimização, contudo sem sucesso: o resultado que obtivemos ficou ao nível da compilação com -O2 obtido pelo gcc 4.4.5. Eventualmente tal justificará a razão pela qual a compilação *default* também não o faz. Ainda em relação a esta técnica, queremos salientar que também o GCC prevê a sua utilização através das flags -fwhole-program –combine. No entanto o manual refere que a opção não é suportada para aplicações FORTRAN. Utilizando os números apresentados pela tabela 4.1, podemos concluir que o SANDER é, na sua maioria, código FORTRAN (99,91%), representando uns expressivos 76,70% do código total (excluindo as AMBERTOOLS); sendo assim, consideramos um exercício desinteressante tentar aplicar uma técnica que, na melhor das hipóteses, apenas considerará 29 fontes C/C++, correspondendo a menos de 4% do código do AMBER[9].

Por último, fica a dúvida em relação a qual seria comportamento da aplicação compilada com as ferramentas Intel, se o processador que utilizámos fosse também ele um Intel [21].

5.5 Optimização Guiada

Os compiladores mais frequentemente utilizados (GCC, Intel C/C++ Compiler, etc.) suportam uma técnica que consiste em compilar uma aplicação de forma que esta, ao ser executada, produza um conjunto de informação – não disponível em tempo de compilação[10] – que provavelmente permitirá ao compilador melhor realizar a tarefa de optimização, produzindo código mais adaptado à realidade.

O argumento facilmente nos seduz, contudo parece-nos evidente que esta técnica se encontra limitada pelo facto de o código gerado ter uma associação directa com a execução, ou *run*, que gerou a informação para o compilador; tal execução deve representar o típico cenário de uso – que pode, todavia, ser demasiado abrangente – o que maximizará a utilidade da informação obtida. Por essa razão, ao aplicarmos esta estratégia poderá ser pertinente realizar uma análise com o objectivo de determinar quais as partes da aplicação a sujeitar ao processo, e qual o caso de uso a executar.

No que se refere ao AMBER, e a este trabalho em particular, a selecção do código fonte não é uma opção, dada a complexidade e dimensão do código; além disso, tal violaria o propósito de apenas considerar oportunidades de optimização não invasivas e, idealmente, fáceis de explorar.

Ainda, nos moldes actuais, esta técnica não encontra aplicação em programas MPI, factor que não antecipámos não obstante a sua exploração ter surgido por curiosidade. Nomeadamente, a utilidade da informação que possamos recolher esgota-se a partir do momento em que fazemos variar o número de processos MPI que lançamos; as estatísticas recolhidas serão

[9] De novo excluindo as AMBERTOOLS.

[10] Por exemplo: quais as funções mais utilizadas ou quais os ramos normalmente seguidos durante a execução (*Branch prediction*), quais os dados mais acedidos, informação estatística sobre o valor das variáveis, etc.

obviamente função desse número[11]. O leitor poderia prontamente indicar uma solução para o problema: correr a aplicação sempre com o mesmo número de processos utilizado durante o primeiro *run*. Tal poderia constituir uma solução, não fosse a existência de outro factor que somos forçados a considerar: não nos é possível determinar onde cada imagem colocará a informação que recolherá, o que significa que existindo vários processos MPI a correr o mesmo programa, ocorrerá uma sobreposição de informação na directoria especificada como *output* para o *profiling*. Portanto mesmo que pudéssemos realizar uma segunda compilação para cada processo, usando a informação por si gerada, não o poderíamos fazer porque não temos forma de dispor dessa informação.

5.6 Infra-estruturas de Rede

Como pudémos concluir anteriormente aquando do profiling que realizámos na secção 4.2, o tempo de execução das operações MPI representa aproximadamente 50% do tempo de total de execução. Por conseguinte, os *interconnects* terão assumidamente um peso determinante na performance que conseguimos obter com a aplicação que temos vindo a usar. De facto, considerando a expressão das comunicações no tempo de execução, esperamos obter uma melhoria de 0,5% no tempo de execução por cada melhoria de um ponto percentual no *interconnect*[12]. Com tal motivação, iniciamos o estudo dos *interconnects*. Iremos considerar as mesmas quatro opções que anteriormente: *InfiniBand* nativo (IB), *IP over InfiniBand* (IPoIB), *IP over Ethernet* (IPoE) e *Myrinet Express over Ethernet* (MXoE). Adicionalmente, exploraremos com menor ênfase a utilização da memória partilhada. Reiteramos que sempre que sinalizarmos a utilização de uma das mencionadas opções, e a não ser que algo em contrário seja dito, estaremos a forçar a utilização exclusiva dessa mesma opção.

Interessa contudo referir que os valores obtidos nos testes anteriores são reflexo da utilização de *InfiniBand* nativo (IB), sem contudo forçar o seu uso exclusivo; utilizámo-lo pois antecipávamos ser a opção que tornava viável a realização de estudos com vários *runs*, como aquele realizado para o GCC[13]. Os resultados que apresentaremos oportunamente confirmam que teríamos em boa medida comprometido os testes que pretendiamos efectuar caso não o tivéssemos feito dadas as limitações de tempo; as conclusões que retiramos não são de forma alguma afectadas por tal opção.

Ao longo das próximas duas subsecções utilizaremos um conjunto de imagens obtidas com a ferramenta de monitorização MUNIN[14]; tais capturas são bastante elucidativas, pelo que optaremos por apenas tecer pequenos comentários muito precisos, quando tal se justificar.

5.6.1 Testes Intra-nó

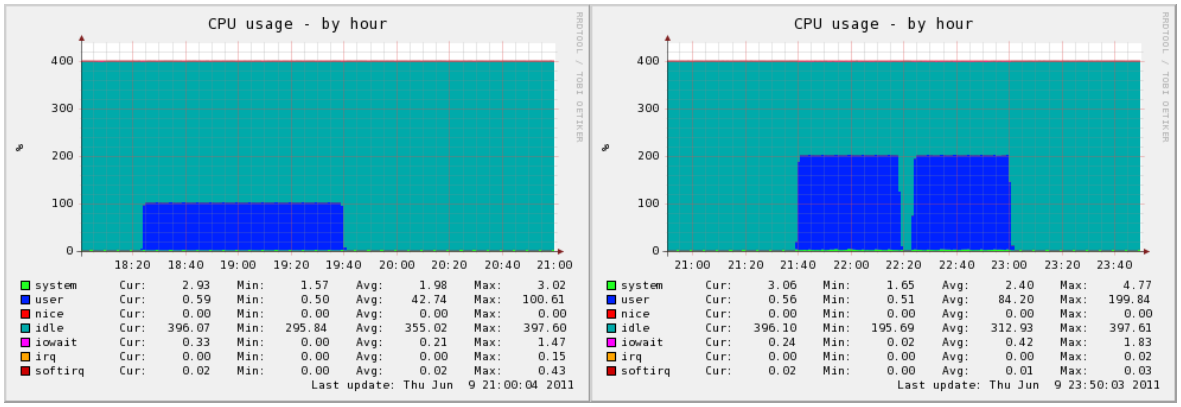
Apresentamos os resultados para a execução com um número NP de processos, NP = 1,2,4, correndo na mesma máquina (intra-nó). Os testes seguintes foram executados especificando `--mca btl sm,self`, o que implica o uso de memória partilhada.

[11] E não só. Muitas aplicações MPI adoptam o modelo *master/slave* e nestes casos a distribuição de trabalho pode ser dinâmica – e frequentemente é – e, por conseguinte, pode variar em função de inúmeros factores não necessariamente constantes ao longo de sucessivas execuções.

[12] Obviamente não podemos contudo assumir que tal seja estritamente linear.

[13] Não é demais relembrar que foi esse estudo que nos permitiu obter aproximadamente 5% de performance extra.

[14] `http://munin-monitoring.org/`



(a) NP = 1 (b) NP = 2

Figura 5.1: Utilização do processador para NP = {1, 2}. Porção correspondente à utilização em *kernel mode* — dada pelo somatório dos máximos de *system*, *irq* e *softirq* — é (ainda) reduzida. Para NP = 2, a zona azul mais à esquerda corresponde à execução no mesmo processador, conforme sugerido pela figura 5.3.

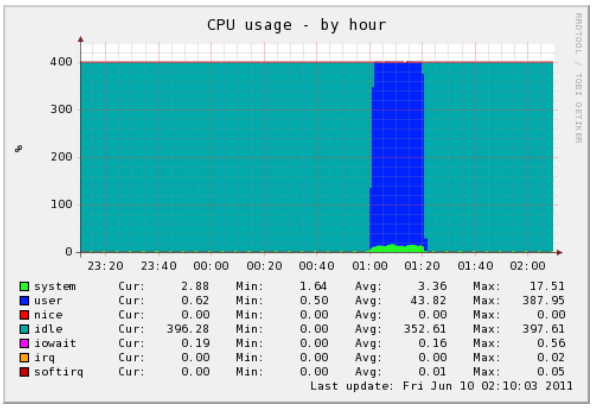


Figura 5.2: Utilização do processador para NP = 4. Zona a verde intenso, correspondendo a aproximadamente 18% de *cpu usage*, indica a execução de código em *system mode*. Ao ter todos os *cores* ocupados com a simulação, o tempo dispendido em trocas de contexto implica necessariamente uma redução no tempo disponível para executar a aplicação. Tal justificará a razão pela qual não obtivemos um *speedup* mais próximo do máximo teórico de 4×.

Tabela 5.10: Tempos para a melhor configuração com gcc-4.4.5 em Intra-nó.

NP	Tempo (s)	*Speedup* (×)
1	4556,858	---
2 (mesmo cpu)	2320,137	1,964
2 (cpus #s)	2226,498	2,047
4	1213,802	3,754

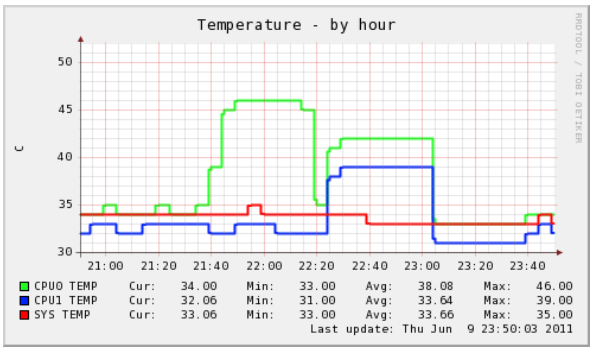
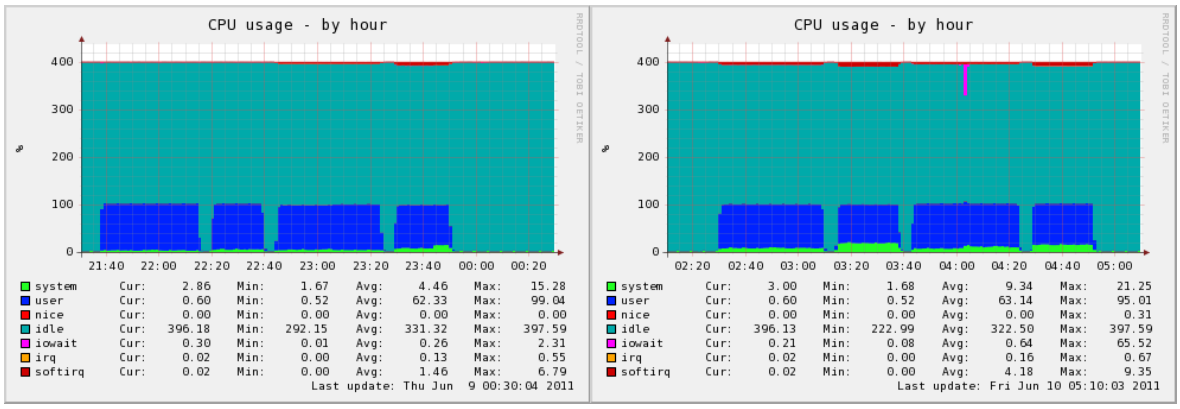


Figura 5.3: Temperatura dos processadores para NP = 2 permite inferir onde os processos estão a correr.

5.6.2 Testes Inter-nó

5.6.2.1 Sem memória partilhada

Os testes aqui apresentados envolvem a distribuição dos processos pelos nós, de forma a que nenhum nó execute mais do que um processo[15]. Por essa razão, não activámos o uso de memória partilhada junto do Open MPI.

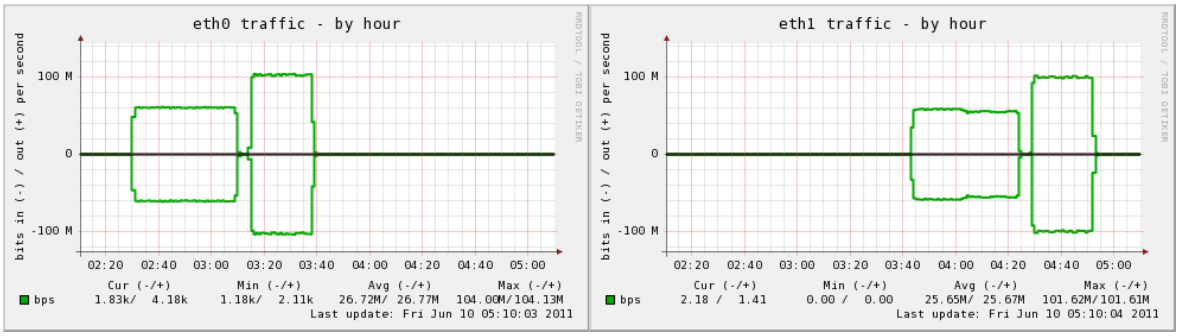


(a) IB e IPoIB (b) IPoE e MXoE

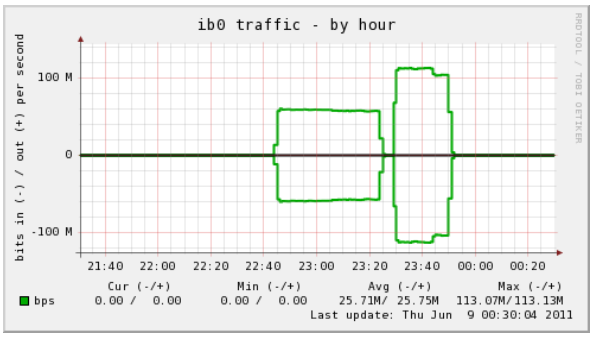
Figura 5.4: Utilização do processador para NP = {2, 4}, com *InfiniBand* (IB), *IP over InfiniBand* (IPoIB), *IP over Ethernet* (IPoE) e *Myrinet Express over Ethernet* (MXoE). Da esquerda para a direita: IB (NP = 2), IB (NP = 4), IPoIB (NP = 2), . . . , MXoE (NP = 4).

É visível na figura 5.4a um aumento no consumo de CPU em *kernel mode* ao passar de IB para IPoIB; nesse contexto, não deve ser ignorado o valor respeitante aos *software interrupts* (*softirq*) – que aliás regista um crescimento ao adoptar IPoIB – e que atinge uns consideráveis 6,79% para NP = 4. Portanto, a utilização total de CPU em *kernel mode* ascende aos 22,62%.

[15]Tal também implica que o número de nós participantes será igual a NP.

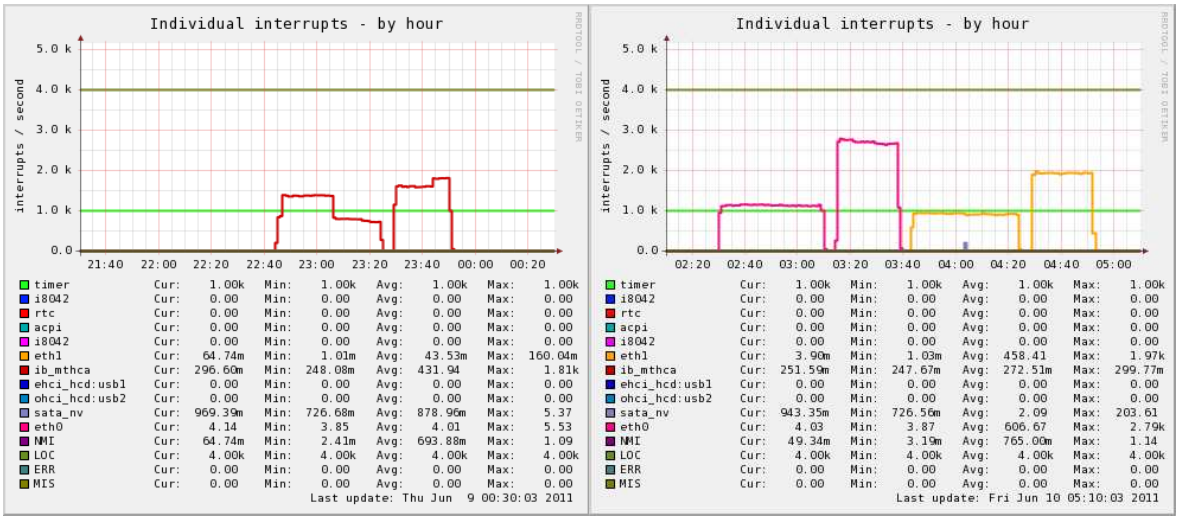


(a) eth0 (b) eth1



(c) ib0

Figura 5.5: Utilização das interfaces de rede ib0 (IPoIB), eth0 (IPoE) e eth1 (MXoE), para NP = {2, 4}.



(a) IB e IPoIB (b) IPoE e MXoE

Figura 5.6: Interrupções geradas por segundo para NP = {2, 4}, com *InfiniBand* (IB), *IP over InfiniBand* (IPoIB), *IP over Ethernet* (IPoE) e *Myrinet Express over Ethernet* (MXoE).

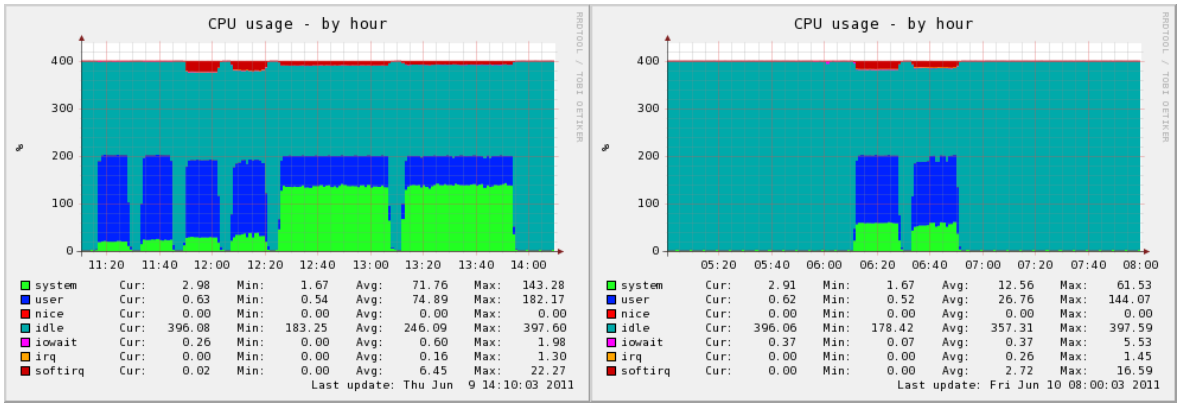
Observa-se assim que o investimento em *hardware* caro (adaptadores IB) se traduz numa redução no uso de CPU quando este (IB) é usado no modo nativo; já o IPoIB, que realiza o *stack* IP à custa do CPU sem aproveitar as possibilidades do *hardware*, comporta-se como as interfaces "*dumb*" GbE.

Tabela 5.11: Tempos para a melhor configuração com gcc-4.4.5 em Inter-nó. *Speedups* em relação a IPoE, para o NP considerado. Verificamos que as diferenças se acentuam à medida que aumentamos o número de processos.

Opção		Tempo (s)	*Speedup* (×)
NP = 2	IPoE	2383,653	---
	MXoE	2435,329	0,979
	IPoIB	2373,463	1,004
	IB	2232,843	1,068
NP = 4	IPoE	1409,7	---
	MXoE	1392,438	1,012
	IPoIB	1269,534	1,110
	IB	1172,308	1,202

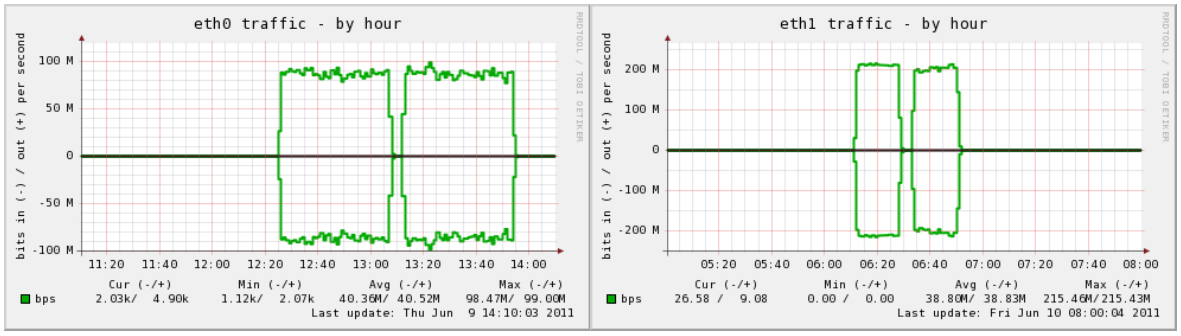
5.6.2.2 Com memória partilhada

Os testes seguintes contam com a activação da memória partilha, devido à execução de múltiplos processos em cada nó.



(a) IB, IPoIB e IPoE (b) MXoE

Figura 5.7: Utilização do processador para NP = 8, com *InfiniBand* (IB), *IP over InfiniBand* (IPoIB), *IP over Ethernet* (IPoE) e *Myrinet Express over Ethernet* (MXoE). Da esquerda para a direita: IB e dois processos no mesmo CPU; IB e dois processos em CPUs distintos; IPoIB e dois processos no mesmo CPU; . . . ; MXoE e dois processos em CPUs distintos. A melhoria introduzida por MXoE face a IPoE é muito significativa



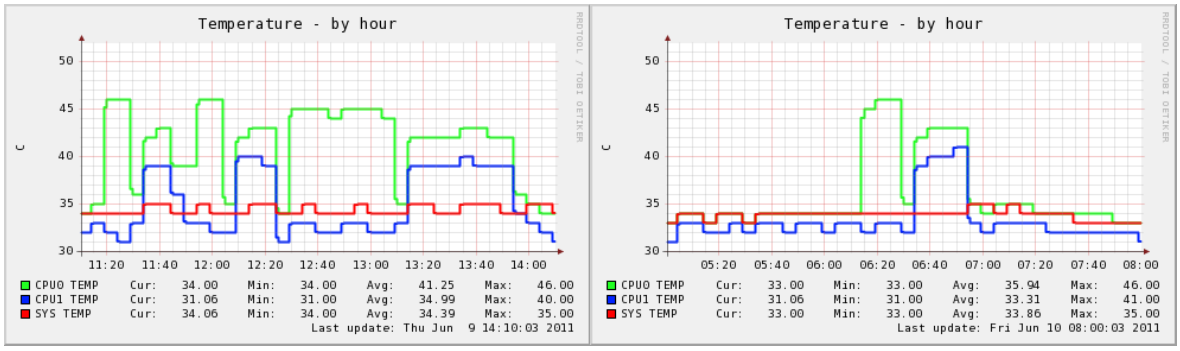
(a) eth0 (b) eth1

(c) ib0

Figura 5.8: Utilização das interfaces de rede ib0 (IPoIB), eth0 (IPoE) e eth1 (MXoE), para NP = 8.

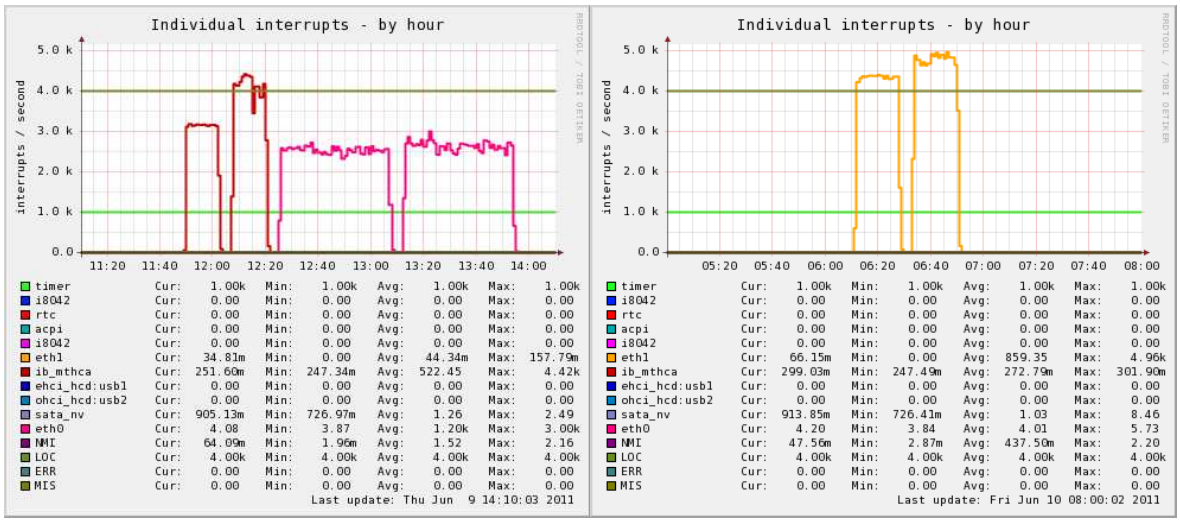
Tabela 5.12: Tempos para a melhor configuração com gcc-4.4.5 em Inter-nó para NP = 8. *Speedups* em relação a IPoE, para o *bidding* considerado. Verificamos que executar os processos no mesmo CPU é a melhor opção; IPoE torna-se proibitivo.

Opção		Tempo (s)	*Speedup* (×)
mesmo CPU	IPoE	2507,081	---
	MXoE	996,822	2,515
	IPoIB	758,526	3,305
	IB	690,044	3,633
CPUs #s	IPoE	2505,098	---
	MXoE	1041,126	2,406
	IPoIB	783,88	3,196
	IB	687,025	3,646



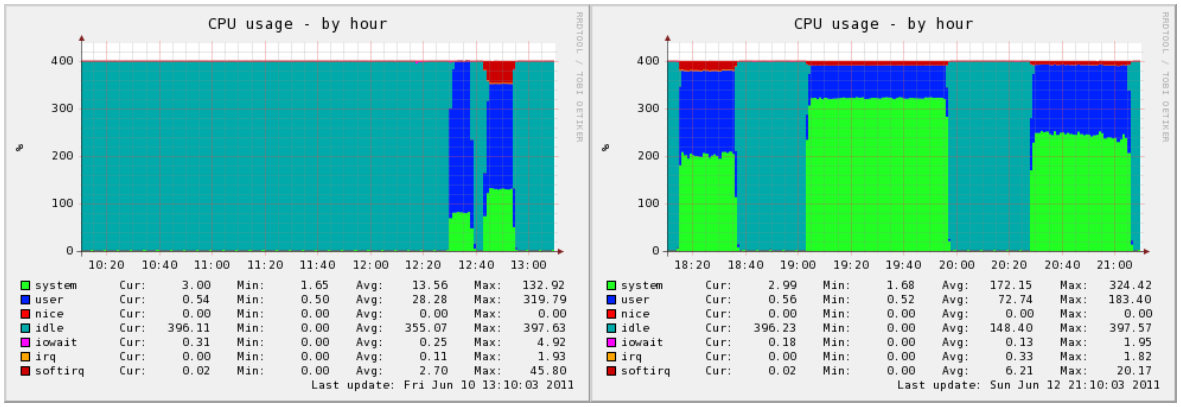
(a) IB, IPoIB e IPoE (b) MXoE

Figura 5.9: Temperatura dos processadores para NP = 8 denuncia quais estão em utilização.



(a) IB, IPoIB e IPoE (b) MXoE

Figura 5.10: Interrupções geradas por segundo para NP = 8, com *InfiniBand* (IB), *IP over InfiniBand* (IPoIB), *IP over Ethernet* (IPoE) e *Myrinet Express over Ethernet* (MXoE).

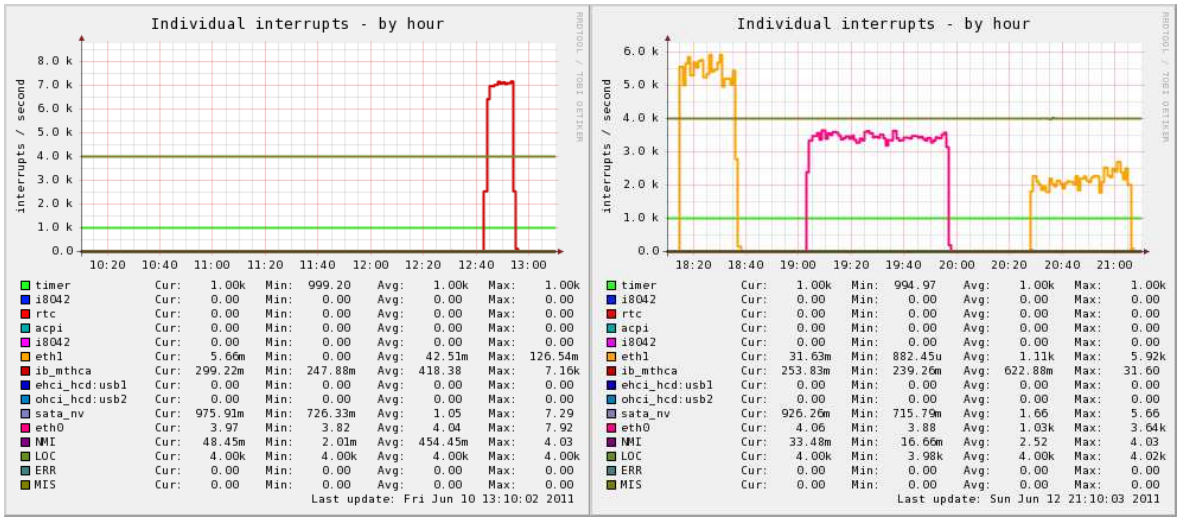


(a) IB e IPoIB (b) MXoE, IPoE e escolha do Open MPI

Figura 5.11: Utilização do processador para NP = 16, com *InfiniBand* (IB), *IP over InfiniBand* (IPoIB), *IP over Ethernet* (IPoE), *Myrinet Express over Ethernet* (MXoE) e a escolha do Open MPI (correspondendo à ausência de especificação do canal a usar).

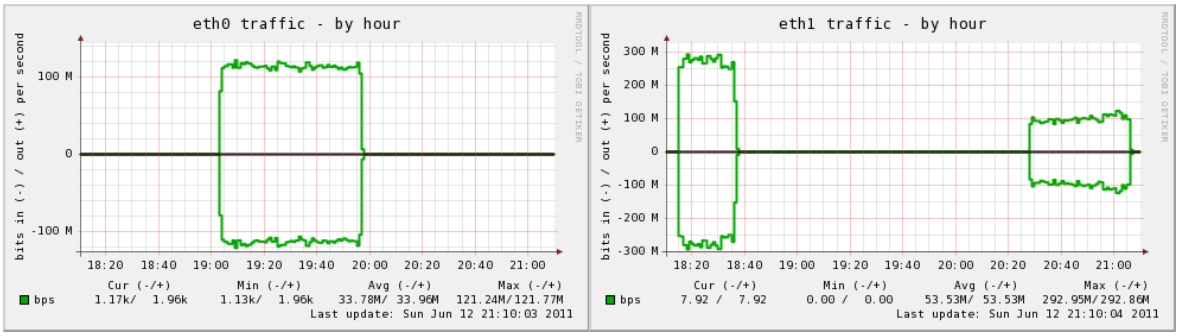
Tabela 5.13: Tempos para a melhor configuração com gcc-4.4.5 em Inter-nó para NP = 16. *Speedups* em relação a IPoE.

Opção	Tempo (s)	*Speedup* (×)
IPoE	3220,311	---
MXoE (usecs = 10 μs)	1242,066	2,593
IPoIB	645,504	4,989
IB	485,965	6,627
sem indicar	2258,538	1,426

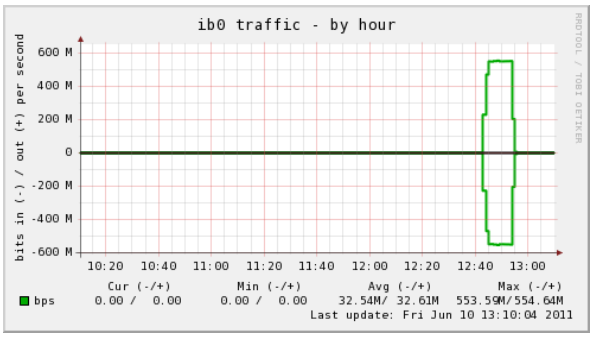


(a) IB e IPoIB (b) MXoE, IPoE e escolha do Open MPI

Figura 5.13: Interrupções geradas por segundo para NP = 8, com *InfiniBand* (IB), *IP over InfiniBand* (IPoIB), *IP over Ethernet* (IPoE), *Myrinet Express over Ethernet* (MXoE) e a escolha do Open MPI.



(a) eth0 (b) eth1



(c) ib0

Figura 5.12: Utilização das interfaces de rede ib0 (IPoIB), eth0 (IPoE) e eth1 (MXoE), para NP = 16.

Tabela 5.14: Tempos para a melhor configuração com gcc-4.4.5 em Inter-nó para NP = 16, com MXoE e em função de rx-usecs. Verifica-se que a redução da latência possibilita um ganho de desempenho comparável com resultados anteriores; a melhoria face à *baseline* eleva-se em consequência.

rx-usecs (μs)	Tempo (s)	*Speedup* (×)
20	1294,837	---
15	1303,18	0,994
10	1242,066	1,042
5	2460,231	0,526
1	3531,123	0,367

Atendendo aos valores patentes na tabela 5.13 somos levados a acreditar que não podemos, de maneira alguma, delegar ao Open MPI a responsabilidade de escolher a infra-estrutura a usar, pois comprovamos que a sua escolha não é certamente a mais adequada, não obstante nos permitir um *speedup* de 1.43× em relação a IPoE. No entanto tal não corresponde inteiramente à verdade; fortuitamente verificámos que tal só acontece ao compilar o Open MPI com suporte para Open-MX, por razão que desconhecemos. Todavia, julgamos importante mostrar ao leitor que situações anómalas acontecem, o que justifica termos mantido o resultado em questão; e que ter uma noção do resultado que devemos obter revela-se fundamental. Outro aspecto de grande relevância é o facto de termos conseguido obter um *speedup* de 2.6× com MXoE face à configuração IPoE *default*, o que é muito bom – em especial se considerarmos que apenas nos exigiu a instalação de software disponibilizado gratuitamente – e certamente importante quando a prioridade seja a utilização, ou construção, de clusters de baixo custo.

5.7 *Streaming SIMD Extensions* (SSE)

Neste capítulo investigamos de que forma podemos beneficiar das extensões SSE. A atenção recai sobre distinguir a utilização de instruções SSE escalares, pouco relevantes, da utilização de instruções SSE vectoriais. Recorde-se que o comportamento SIMD apenas se obtém em consequência do uso de instruções vectoriais que operam sobre um conjunto de bits de uma só vez.

Por conseguinte, estamos interessados em saber que instruções estão presentes nos executáveis da aplicação SANDER. Para o efeito, iremos "desassemblar" os seus executáveis – recorrendo ao `objdump`[16] para expor as instruções máquina – e seguiremos uma abordagem que consiste em procurar por um conjunto restrito de instruções conhecidas, concretamente as correspondentes às quatro operações aritméticas básicas. Tal dar-nos-á uma boa noção da realidade.

Convém observar que as instruções podem ser de precisão simples ou dupla. Sendo assim, procuraremos pelas instruções presentes na tabela seguinte:

Tipo	Instruções
Packed double	addpd,divpd,mulpd,subpd
Packed single	addps,divps,mulps,subps
Scalar double	addsd,divsd,mulsd,subsd
Scalar single	addss,divss,mulss,subss

5.7.1 SSE com gcc-4.4.5

Em primeiro, procurámos compilar o AMBER sem a utilização de extensões SSE, de forma a perceber qual a diferença ao nível da performance. Essa possibilidade é-nos oferecida pelo seu *script* de configuração[16]. Se assim o fizermos, o novo ficheiro de configuração verá suprimidas as referências a "-mtune=generic", as quais estão presentes na configuração *default*. No entanto, tal revela-se incapaz de prevenir a utilização das instruções em causa[17].

De facto, estando a gerar código de 64 bits a utilização de instruções SSE encontra-se activa por omissão, segundo o manual, pelo que para ter a certeza que nada do set SSE é utilizado deveremos passar ao gcc as flags "-mfpmath=387 -mno-sse -mno-sse2". Contudo, não foi possível compilar o AMBER dessa forma, já que a compilação termina precocemente devido a um erro numa fonte FORTRAN[18]; suspeita-se que o código inclui a utilização explicita de instruções do *set*. Infelizmente os conhecimentos do autor nessa linguagem são insuficientes para realizar esta análise.

Para já, concentremo-nos na compilação *default*. Para a aplicação sequencial `sander`, encontrámos a utilização de 55278 instruções escalares e apenas 1326 vectoriais. No caso da versão paralela, `sander.MPI`, o cenário não é muito diferente: 59601 instruções escalares e somente 1432 vectoriais.

Estes números não têm de ser maus, por si. Podem significar uma de duas coisas: a capacidade de vectorização do GCC é medíocre, e nesse caso única possibilidade é testar com um outro compilador para possivelmente aumentar a nossa convicção que assim o é; ou, de facto, o código não permite fazer muito melhor.

Procurando perceber se poderíamos melhorar a expressão das instruções vectoriais sem contudo nos afastarmos demasiado da compilação *default*, procedemos à afinação das opções

[16] Deve ser passada a opção -nosse.

[17] Verificou-se que o número de instruções encontradas neste caso é de resto igual ao que indicaremos oportunamente para a compilação *default*, o que corrobora a improficuidade do *switch* -nosse.

[18] _nose_hoover.f:123: error: SSE register return with SSE disabled.

default, concretamente substituindo `generic` por `opteron` (i.e., usando -mtune=opteron). Com esta mudança esperaríamos melhorar a taxa de utilização de instruções vectoriais, o que marginalmente conseguimos: 55294 instruções escalares e 1331 vectoriais, para a versão sequencial; 59612 instruções escalares e 1437 instruções vectoriais para a versão paralela.

A tabela seguinte resume estes, e outros indicadores.

Tabela 5.15: A grande conclusão é que as alterações são mínimas entre as versões testadas, o que se traduz em mudanças muito ténues no rácio entre o número de instruções vectoriais (*packed*) e o número de instruções escalares (*scalar*).

Opção		*Packed*		*Scalar*		Rácio	Tempo (s)
		double	*single*	*double*	*single*		
sander	default	1326	0	55247	31	0,02399	4449,145
	opteron	1331	0	55263	31	0,02407	4382,294
	melhor	1331	0	55199	31	0,02410	4273,333
sander.MPI	default	1432	0	59565	36	0,02403	507,514
	opteron	1437	0	59576	36	0,02411	499,376
	melhor	1441	0	59486	36	0,02421	485,965

Não podemos ignorar que, apesar de tudo, conseguimos uma redução do tempo de execução.

5.7.2 SSE com Intel

O resultado obtido com a compilação *default* em Intel parece indicar que a capacidade de vectorização dos seus compiladores excede consideravelmente aquela demonstrada previamente pelo gcc-4.4.5, de acordo com os resultados que obtivemos: 66627 instruções escalares e 18091 instruções vectoriais para a versão sequencial; 71647 escalares e 19319 vectoriais para a versão paralela.

Estando a compilar com Intel, entendemos adequado explorar uma opção oferecida pela configuração do AMBER, que apenas se aplica nesta situação: a definição de uma variável de ambiente, SSE_TYPES=*code*, que implica a adição da flag -ax*code*. Para o sistema em causa, queremos exportar SSE_TYPES=SSE2, anteriormente à configuração, sinalizando que pretendemos a geração de instruções SSE e SSE2[19], conforme descreve o manual. Verificámos, porém, que tal acção não tem qualquer efeito sobre o número de instruções que contabilizamos. A tabela 5.16 condensa a informação obtida.

Note-se que o número de instruções vectoriais é mais de 14 vezes o obtido com o gcc-4.4.5, e que efectivamente se aumenta o rácio entre as instruções vectoriais e escalares. No entanto, da mesma forma que o resultado anterior não era necessariamente mau, este não é necessariamente bom. Ao incrementar de forma tão expressiva o número de instruções SSE vectoriais, suporíamos uma performance bem melhor que os tímidos 5,5% obtidos face ao gcc-4.4.5, na configuração *default*[20] (e não estamos sequer a considerar que a compilação Intel tem a vantagem de recorrer a uma forma de IPO). Assim sendo, e não obstante não sermos conhecedores da totalidade do código – o que pode comprometer a nossa análise – ficamos com a sensação que a elevada taxa de vectorização dos compiladores Intel será obtida à custa da geração de código *dummy*. O facto de não se verificar uma redução do número de instruções escalares assim o sugere: de facto, gostaríamos de ter visto o rácio mencionado

[19] Infelizmente não foi possível realizar o procedimento exactamente como descrevemos, visto termos obtido um erro de compilação ao qual somos alheios. Verificámos contudo que passar a flag -nobintraj evita esse mesmo erro de ocorrer. Lamentavelmente, a simulação que tem vindo a ser utilizada requer a compilação do bintraj, razão que nos levou a passar a flag -ax*code* via AMBERBUILDFLAGS. Desta forma, conseguimos compilar a aplicação com sucesso.

[20] Consultar a tabela 5.9.

aumentar em relação aos testes com gcc-4.4.5 e, em simultâneo, um estreitamento do número de instruções escalares[21]. Tal não acontece[22]

Tabela 5.16: SSE com Intel.

Opção		*Packed*		*Scalar*		Rácio	Tempo (s)
		double	*single*	*double*	*single*		
sander	default	18091	0	66593	34	0,27153	3926,07
	+ '-axSSE2'						3937,766
sander,MPI	default	19307	12	71608	39	0,26964	480,838
	+ '-axSSE2'						476,173

5.8 Ligação Dinâmica vs. Ligação Estática

A tabela 5.17 apresenta os tempos de execução da aplicação `sander.MPI` quando ligada estaticamente[23] e recorda aqueles obtidos com a ligação dinâmica.

Os resultados demonstram que a ligação estática afecta negativamente, embora de forma ligeira, a performance da aplicação. O profiling recorrendo ao CodeAnalyst surge na tabela 5.18 e é bastante relevador da origem dos resultados apresentados. Acontece que somos confrontados pelo aumento drástico (+336,14%) do número de *misses* na cache de instruções, que verá o seu efeito ampliado, acreditamos, pela redução considerável de *hits* na L2 ITLB após *miss* na L1 ITLB (-33,62%).

Tabela 5.17: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado de acordo com a opção indicada. *Speedup* das versões estáticas em relação às dinâmicas.

Opção	Tempo (s)		*Speedup* (×)
	Dinâmica	Estática	
-O3 -mtune=generic	507,514	516,433	0,983
-mtune=opteron -O3 -fno-tree-ter -fno-tree-fre -fno-tree-switch-conversion -fno-delayed-branch -fno-gcse-after-reload	485,965	496,891	0,978

[21] No fundo, tal traduz-se a ideia que uma substituição, ainda que parcial, ocorresse.

[22] Sendo mais preciso, o número de instruções encontradas aumenta aproximadamente 49,67% e 49,04% em relação ao GCC para as versões sequencial e paralela, respectivamente, quando consideramos a compilação *default*.

[23] É necessário realizar a configuração com as opções -static e -noX11. Note-se que tal implicou a instalação dos pacotes de suporte libibverbs_static e mthca_static, em todos os nós, e a recompilação do Open MPI, com ligação estática.

Tabela 5.18: Profiling com AMD CodeAnalyst com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.4.5, de acordo com a melhor configuração determinada.

Evento	**Dinâmica**	**Estática**	
CPU clocks not halted	13572828	13212664	-2,65%
Retired branch instructions	9725846	9356988	-3,79%
Retired mispredicted branch inst.	429558	426912	-0,62%
L2 fill/writeback	1979239	2059003	+4,03%
L2 cache misses	210853	192764	-8,58%
Requests to L2 cache	1833268	1907280	+4,04%
Retired instructions	11736208	11746855	+0,09%
Data cache accesses	5171746	5082558	-1,72%
Data cache misses	1495404	1505370	+0,67%
Data cache refills from L2 or system	1490392	1501482	+0,74%
L1 DTLB miss and L2 DTLB miss	22416	20542	-8,36%
L1 DTLB miss and L2 DTLB hit	1534440	1408518	-8,21%
Misaligned accesses	91050	93074	+2,22%
Retired instructions	11654998	11534588	-1,03%
Instruction cache fetches	4843428	4806158	-0,77%
Instruction cache misses	16416	71596	+336,14%
L1 ITLB miss and L2 ITLB hit	6062	4024	-33,62%
L1 ITLB miss and L2 ITLB miss	88	84	-4,55%
Retired instructions	11629296	11628744	-0,005%

5.9 *Hugepages*

Nesta secção exploramos o potencial benefício inerente à utilização de *hugepages*. Como tivemos oportunidade de explicar, a vantagem da utilização de páginas maiores é a eventual redução do número de misses nas TLBs, o que poderá ter consequências positivas ao nível da performance de uma aplicação, admitimos, se a mencionada redução for suficientemente expressiva. Tendo isto em mente, não apenas realizámos ensaios com gcc-4.4.5 mas também com o seu predecessor, o gcc-4.1.2, visto termos apurado anteriormente que este último apresenta maior número de *misses* nas referidas caches.

A biblioteca libhugetlbfs possibilita-nos a sua utilização de duas formas distintas: recompilando o código da respectiva aplicação de forma a ser ligado à biblioteca em questão, escolhendo entre guardar as secções .bss, .data e .text (respectivamente as variáveis não inicializadas, as inicializadas e o código executável) em *hugepages* (método BDT) ou somente a secção .bss (método B); ou, alternativamente, forçando com que cada chamada `malloc` da aplicação[24] seja substituída[25] pela chamada devida da biblioteca, que tratará que o alocamento de memória seja feito em páginas de 2MB. Esta última dispensa qualquer recompilação da aplicação.

A tabela seguinte apresenta os resultados para os ensaios considerados.

[24] Da **libc** para ser mais correcto.
[25] Bastando exportar `LD_PRELOAD=libhugetlbfs.so` e `HUGETLB_MORECORE=yes`.

Tabela 5.19: Tempos relativos à execução da aplicação sander.MPI com a biblioteca *libhugetlbfs*. *Speedups* em relação aos resultados sem *hugepages*: 555,797 e 485,965 segundos para o gcc-4.1.2 e gcc-4.4.5, respectivamente.

Opção	Método	Tempo (s)		Páginas	*Speedup* (×)
		gcc-4.1.2	gcc-4.4.5		
-O3 -mtune=opteron	*malloc*	564,902	---	105	0,984
melhor	B	---	497,991	184	0,976
	BDT	---	511,056	196	0,951
	malloc	---	520,761	110	0,933

Com estes resultados conclui-se que a utilização de *hugepages* é incapaz de melhorar os resultados obtidos previamente para a aplicação em causa; na verdade, registamos uma perda de performance, que nos melhores casos foi apenas marginal. Tal faz-nos supor que esta técnica poderá ser útil quando confrontados com uma aplicação que apresente um consumo apreciável de memória, o que não é o caso[26]. Procurando conhecer quais as implicações da utilização de páginas maiores ao nível do hardware, realizámos um profiling da aplicação em duas das configurações anteriores. Comecemos pelo gcc-4.1.2, exibindo a tabela que se segue.

Tabela 5.20: Profiling com AMD CodeAnalyst com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.1.2, de acordo com a configuração previamente indicada. Consagra-se a utilização de *hugepages* (método *malloc*).

Evento	**sem**	**com**	
CPU clocks not halted	15012602	15476120	+3,09%
Retired branch instructions	9999660	9953466	-0,46%
Retired mispredicted branch inst.	429938	410498	-4,52%
L2 fill/writeback	2061746	2187030	+6,08%
L2 cache misses	201091	279489	+38,99%
Requests to L2 cache	1889678	2080645	+10,11%
Retired instructions	13286470	13269750	-0,13%
Data cache accesses	6902282	6971932	+1,01%
Data cache misses	1571940	1634544	+3,98%
Data cache refills from L2 or system	1570682	1633990	+4,03%
L1 DTLB miss and L2 DTLB miss	23196	73100	+215,14%
L1 DTLB miss and L2 DTLB hit	1557930	10968	-99,30%
Misaligned accesses	92388	437588	+373,64%
Retired instructions	13293702	13280888	-0,10%
Instruction cache fetches	5522510	5516678	-0,11%
Instruction cache misses	38040	27590	-27,47%
L1 ITLB miss and L2 ITLB hit	5228	5914	+13,12%
L1 ITLB miss and L2 ITLB miss	416	80	-80,77%
Retired instructions	13342072	13263930	-0,59%

[26]O leitor poderá averiguar que a utilização máxima de memória se situa nos 392 megabytes, visto cada página dispor de 2048 kB. Tal representa apenas 10% da memória instalada em cada nó do *cluster*.

Os valores apresentados não deixam margem para dúvidas. Percebemos que a utilização de *hugepages* é positiva para a cache de instruções, reduzindo em mais de 80% os *misses* em simultâneo na L1 ITLB e L2 ITBL, ao mesmo tempo que aumenta em 13,12% os *hits* na L2 ITLB após *miss* na L1 ITLB; lamentavelmente os acessos à cache de dados saem prejudicados, verificando-se um acréscimo explosivo de 215% nos *misses* em ambas as ITLBs, acompanhado de uma redução muito significativa dos *hits* na L2 ITLB após *miss* na L1 ITLB (-99,30%). Observamos ainda uma subida abrupta do número de acessos a dados mal alinhados (+373,64%).

A ideia é portanto que neste caso em concreto a utilização da biblioteca *libhugetlbfs* retira o problema de um lado para o colocar noutro, o que suspeitamos fará com que o resultado final seja apenas ligeiramente negativo não obstante o cenário pouco amistoso que relatámos para os acessos à *data cache*. Antevemos que uma situação semelhante poderá, por conseguinte, ocorrer com o gcc-4.4.5. Eis o profiling para este.

Tabela 5.21: Profiling com AMD CodeAnalyst com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.4.5, de acordo com a configuração previamente indicada. Consagra-se a utilização de *hugepages* (método B).

Evento	**sem**	**com**	
CPU clocks not halted	13572828	13359018	-1,58%
Retired branch instructions	9725846	9750778	+0,26%
Retired mispredicted branch inst.	429558	436462	+1,61%
L2 fill/writeback	1979239	1987406	+0,41%
L2 cache misses	210853	210816	-0,02%
Requests to L2 cache	1833268	1851803	+1,01%
Retired instructions	11736208	11609632	-1,08%
Data cache accesses	5171746	5142760	-0,56%
Data cache misses	1495404	1504912	+0,64%
Data cache refills from L2 or system	1490392	1502622	+0,82%
L1 DTLB miss and L2 DTLB miss	22416	19340	-13,72%
L1 DTLB miss and L2 DTLB hit	1534440	1401994	-8,63%
Misaligned accesses	91050	91890	+0,92%
Retired instructions	11654998	11616776	-0,33%
Instruction cache fetches	4843428	4856990	+0,28%
Instruction cache misses	16416	14734	-10,25%
L1 ITLB miss and L2 ITLB hit	6062	6236	+2,87%
L1 ITLB miss and L2 ITLB miss	88	64	-27,27%
Retired instructions	11629296	11690306	+0,52%

Na verdade os valores apresentados pela tabela 5.21 mostram que ambas as opções desempenham, no geral, bastante próximas uma da outra; são visíveis pontos negativos (compensados, diríamos, por outros positivos) em ambas, que pela sua expressão moderada justificarão o resultado marginalmente inferior obtido com *hugepages* para a configuração em causa. Pelo dito anteriormente, excluímos a utilização de hugepages com o gcc-4.1.2, distribuído com o CentOS 5.5, e reiteramos a nossa convicção que facilmente o resultado final pode ser positivo se em causa estiver a utilização de uma aplicação com um perfil de utilização de memória diferente.

5.10 Intel MKL

A suite de compiladores Intel usada neste trabalho inclui a biblioteca Intel MKL, muito optimizada, que pode ser utilizada pelo AMBER, substituindo as bibliotecas LAPACK e BLAS com este distribuídas[27]. Nesta secção verificamos se a utilização desta biblioteca efectivamente se revela uma mais-valia no que se refere à performance e, adicionalmente, se essa eventual melhoria também acontece se o código for compilado com o GCC.

As tabelas 5.22 e 5.23 exibem os tempos apurados para a execução com a Intel MKL, versão sequencial e compilada estaticamente conforme determinado pela configuração do AMBER[28]. Os resultados demonstram que a utilização da biblioteca mencionada não altera, na prática, a performance do *run*. Por assim ser, especulámos que tais resultados poderiam ser fruto da utilização de uma simulação relativamente rápida de concluir[29]. Por essa razão, optou-se por voltar a testar a utilização da MKL, desta feita recorrendo à simulação *Cellulose_production_NVT* da mesma *suite* de testes de onde obtivemos a simulação usada até então, porém computacionalmente muito mais exigente que a anterior[30]. Não obstante, o cenário manteve-se inalterado. Por conseguinte, concluímos que a utilização da biblioteca Intel MKL não é, no que diz respeito ao AMBER, de considerar[31].

Tal não significa, de forma alguma, que as ferramentas Intel não sejam uma opção a considerar noutros contextos.

Tabela 5.22: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com Intel, de acordo com a opção indicada. *Speedup* em relação ao Amber *default* com Intel.

Opção	Tempo (s)	*Speedup* (×)
Amber 11 *default*	480,838	---
+ Intel MKL	476,442	1,009

Tabela 5.23: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando *InfiniBand*. AMBER compilado com gcc-4.4.5, de acordo com a opção indicada. *Speedup* em relação à melhor configuração determinada.

Opção	gcc (s)	*Speedup* (×)
-mtune=opteron -O3 -fno-tree-ter -fno-tree-fre -fno-tree-switch-conversion -fno-delayed-branch -fno-gcse-after-reload	485,965	---
+ Intel MKL	490,445	0,991

[27] Encontram-se respectivamente em `.../AmberTools/src/lapack` e `.../AmberTools/src/blas`.

[28] As bibliotecas LAPACK e BLAS originais são também ligadas estaticamente, pelo que a coerência é mantida. Sobre esta questão importa esclarecer que foram tentadas outras configurações, nomeadamente a ligação dinâmica da biblioteca e, também, a sua versão paralela. Podemos avançar que os resultados obtidos não diferem dos que apresentaremos de seguida.

[29] Reiterando contudo que foi necessária a utilização um exemplo "curto" para que fosse possível ao autor a experimentação exaustiva.

[30] Os autores descrevem esta simulação como sendo representativa de simulações do "mundo real". De facto demorou-nos aproximadamente 150 minutos, ou seja 2,5 horas, em 16 *cores* com *InfiniBand* nativo.

[31] Frisando que nos é impossível testar todas as facetas da aplicação, com diversos exemplos, pelo que a nossa conclusão é naturalmente limitada. Contudo, julgamos que se a sua utilização não foi significativa com um exemplo dito representativo, dificilmente o será noutro qualquer.

5.11 AMD Open64

Tentámos compilar o AMBER com os compiladores da *suite* Open64[32], porém sem sucesso. Confrontados com a ausência de suporte à *suite* mencionada por parte do *script* de configuração das AMBERTOOLS, vimo-nos forçados a adulterar um conjunto de ficheiros de configuração e a aplicar um *patch* apenas disponível na *mailing list* dos programadores do AMBER[33]. Não obstante, por último deparámo-nos com um erro[34] cuja correcção implicaria, tanto quanto percebemos, alterar o código fonte – tarefa que não consideramos de todo. Note-se que a fonte que causa o erro, `ncsu-cv-DISTANCE.f`, não foi alterada pelo *patch* que aplicámos. Sendo assim, e porque nenhum dos compiladores que anteriormente utilizámos apresentou tal erro, admitimos tratar-se de uma questão de regras que o Open64 observa, contrariamente aos seus homólogos.

5.12 AMD ACML

Não obstante ter-nos sido impossível utilizar com sucesso as ferramentas de compilação Open64 junto do AMBER, fomos capazes de o ligar com a biblioteca AMD ACML, que nos fornece um conjunto de rotinas matemáticas muito optimizadas[35]. Estamos particularmente interessados em averiguar se a utilização desta biblioteca nos confere algum proveito, contrariamente ao que ocorreu com a biblioteca Intel MKL.

Lamentavelmente, apenas podemos apresentar os resultados respeitantes à compilação com gcc-4.1.2; devido um conjunto de erros de compilação que obtivemos ao tentar compilar com o gcc-4.4.5 — e que suspeitamos estarem relacionados com a não exportação de um conjunto de símbolos pela biblioteca `libgfortran`[36] — não estamos em condições de apresentar os resultados referentes à compilação com o mesmo. Confrontados com esse facto, ponderámos compilar o AMBER com uma versão mais recente do GCC, especulando que talvez nos solucionasse o problema, porém não o chegámos a fazer; sentimos que tal esforço seria mal recompensado, admitindo que o conseguíssemos, já que deixariamos de ter uma referencia que nos possibilitasse avaliar, comparativamente, os resultados que obteríamos. Posto isto, a tabela seguinte sumariza as conclusões que pudemos retirar.

Tabela 5.24: Tempos de execução com a aplicação `sander.MPI` (NP = 16), utilizando a opção indicada. AMBER compilado com gcc-4.1.2, de acordo com a configuração *default* e consagrando a utilização da biblioteca AMD ACML. *Speedup* em relação à compilação sem a referida biblioteca.

Opção	Tempo (s)		*Speedup* (×)
	sem ACML	com ACML	
IPoE	3425,29	3307,634	1,035
IB	558,689	553,871	1,009

A utilização da biblioteca AMD ACML não consegue cativar-nos, ainda que nos conduza a melhores resultados quando comparados com aqueles obtidos pela Intel MKL — de certa forma esperados, já que a biblioteca em questão é fornecida pelo fabricante dos processadores

[32] Consultar `http://developer.amd.com/tools/open64` para mais informações.

[33] `http://archive.ambermd.org/201105/0681.html`

[34] *The overall size of the dummy argument array is greater than the size of this actual argument.*

[35] Novamente, registamos a ausência de suporte para tal biblioteca da parte do AMBER. Por conseguinte, foi necessário proceder a algumas alterações à configuração, de modo a não serem utilizadas as implementações BLAS e LAPACK que vêm de origem.

[36] Uma situação semelhante afectou recentemente uma versão ulterior à que usamos: `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=47757`.

que equipam o *cluster* que utilizamos. Sai reforçada a ideia que o desempenho da aplicação `sander.MPI` é bastante independente da presumível qualidade das rotinas matemáticas que utiliza.

5.13 Execução em SMP

Até agora temos circunscrito todos os nossos passos à execução da aplicação SANDER no *cluster* em questão, antevendo que será esta a arquitectura onde tipicamente a anterior executará. No entanto, a utilização de uma única máquina multi-processador é inteiramente legitima; dadas as suas características especiais, brevemente afloradas na secção 3.2, a experimentação com uma máquina cc-NUMA torna-se atraente no âmbito deste trabalho.

Assumidamente não consideramos explorar em profundidade todos os aspectos que analisámos até então para o nosso cluster[37]; em vez disso, estamos particularmente empenhados em averiguar de que forma as diferentes distancias entre processadores (*hop count*) afectam a performance da aplicação e, se adequado, sugerir ao utilizador uma forma de obter a melhor performance possível[38]. O factor escalabilidade, muito pertinente neste contexto, será também observado. Para recordar as características da máquina usada, o leitor deverá revisitar a secção 3.2.

Por conseguinte, começamos por realizar um conjunto de testes em paralelo colocando dois processos (NP = 2) em processadores com diferentes *hops* entre si (1,2,3), esperando observar o efeito inerente à topologia.

Para nos ser possível controlar quais os *cores* disponíveis para executar a aplicação, bem como impor que em cada um desses apenas executasse um processo, recorremos à aplicação `taskset` e passamos os argumentos apropriados ao `mpirun`. Sendo assim, para executar uma aplicação paralela MPI nos cores 1 e 2, faríamos:

```
mpirun -np 2 -bind-to-socket -bysocket taskset -c 1,2 ...
```

Sobre isto, e em relação à aplicação usada (e respectiva simulação), importa mencionar que observámos que o comando indicado faz com que os processos se mantenham em *cores* diferentes durante a execução e com que a memória desça em ambos os nós, como esperaríamos que acontecesse.

A tabela 5.25 mostra os tempos que registámos para os diferentes cenários.

Tabela 5.25: Tempos de execução com a aplicação `sander.MPI` no SMP (2.8 GHz). AMBER compilado com gcc-4.4.5, de acordo com a melhor configuração determinada.

CPUs	*Cores*	*Hops*	Tempo (s)	*Speedup* (×)
0,1	0,2	1	2102,83	---
0,6	0,12	2	2117,04	0,993
0,7	0,14	3	2105,04	0,999

Da sua análise concluímos que para NP = 2 o rácio entre os tempos de comunicação e computação será reduzido, o que justificará as diferenças marginais observadas para diferentes *hops*. Tal não nos surpreende grandemente; de facto, já tínhamos observado aquando da realização do estudo dos *interconnects* que o peso da comunicação, inferido pela utilização

[37] Devido a restrições temporais.

[38] Facilmente se percebe que tal só se aplica quando for utilizado um numero de processadores inferior ao número de processadores disponíveis.

do CPU em *kernel mode*, aumenta consideravelmente quando o número de processos duplica. Simplesmente, dada a topologia seria impossível realizar um ensaio com NP = 4 (ou maior) que apenas utilizasse uma das distancias existentes e em que todos os processos corressem num processador[39]. Mesmo que arranjássemos alguma configuração aceitável, ao aumentar o número de processadores participantes correríamos o risco de algum desses processadores estar a executar alguma aplicação ou serviço a correr no sistema, deturpando o resultado.

No fundo percebemos que é difícil obter valores totalmente credíveis, a não ser que controlássemos efectivamente todos os processos do sistema. Tal não é viável, visto não dispormos de uso exclusivo do equipamento, situação com que a maioria dos utilizadores se deparará.

Apresentamos agora os resultados dos testes de escalabilidade, correspondendo à execução para NP = 1,2,4,8 e 16. Note-se que para estes testes não se impôs qualquer restrição sobre processadores a usar, ou à distribuição dos processos pelos mesmos. Dada a variação que pode ocorrer realizámos dois ensaios, admitindo que nos possam dar mais indicações relativamente à questão anterior.

Tabela 5.26: Tempos de execução com a aplicação `sander.MPI` no SMP (2.8 GHz). AMBER compilado com gcc-4.4.5, de acordo com a melhor configuração determinada. *Speedup* em relação a NP = 1, utilizando o melhor ensaio.

NP	Tempo (s)		Variação (%)	*Speedup* (×)
	ensaio 1	ensaio 2		
1	4950,16	5157,55	+4,19	---
2	2187,27	2192,28	+0,23	2,263
4	2143,13	1177,68	-45,05	4,203
8	890,68	1182,69	+32,79	5,558
16	538,31	440,17	-18,23	11,246
Cluster, NP = 16, IB	485,965	---	---	---

De facto dão. A tabela 5.26 mostra que o resultado apresenta grande variabilidade para NP > 2, que pela sua expressão será atribuível à comunicação entre cpus com diferentes *hops*. Percebemos então que um utilizador pode ver o seu trabalho demorar até duas vezes mais do que o esperado se perante uma distribuição assumidamente infeliz. Todavia, a expressão do problema reduz-se à medida que NP cresce; admitimos que a justificação seja que, a partir de certo ponto, a probabilidade de não utilizar todas as distâncias seja cada vez menor, o que implicará uma menor capacidade de alterar o resultado.

Atendendo ao caso aparentemente mais problemático, correspondendo a NP = 4, podemos afirmar que conseguimos melhorar o resultado, em mais de 17%, ao confinar os 4 processos a 2 processadores com um número de *hops* igual a 1 entre si (CPUs 1 e 3). Admitimos que ao monopolizar um CPU, o escalonador do *kernel* será inteligente o suficiente para não atribuir esse processador a outros processos, caso contrário provavelmente o desempenho seria pior.

Sendo assim, sugerimos ao leitor que tente encontrar um conjunto de cpus que minimizem as distâncias entre si, observando a topologia da máquina. Alternativamente poderá isolar um conjunto de cpus para executar a sua aplicação. Tal implica inicializar o kernel com a opção `isolcpus= cpu_number [, cpu_number ,...]` e posteriormente recorrer ao `taskset` para utilizar os cpus indicados anteriormente. Infelizmente tal requer acesso privilegiado ao sistema.

[39] O intuito de correr todos os processos num CPU diferente é permitir a obtenção de um extremo quando a pior distancia fosse utilizada, possibilitando a comparação entre o melhor e o pior caso.

5.14 Execução com CUDA

Nesta secção apresentamos os resultados relativos à execução com CUDA e discutimos, de forma muito superficial, alguns aspectos relacionados com a mesma. A nossa abordagem[40] é deliberadamente minimalista, visto não vislumbrarmos quaisquer hipóteses de alterar o resultado final. Começamos referindo que, no que diz respeito à execução CUDA, o AMBER suporta três "modelos" de precisão: single-precision (SPSP), híbrido ou misto (SPDP) e double-precision (DPDP). Note-se que o modelo misto é escolhido por omissão, acreditando-se ser a opção com melhor relação entre desempenho e precisão[41]. Concretamente, este modelo utiliza precisão simples para os cálculos e precisão dupla para acumulação. Os restantes modelos utilizam somente precisão simples[42] ou dupla para ambas as operações, respectivamente.

A tabela 5.27 permite-nos aferir de que forma a escolha de um dos modelos citados afecta a performance aplicação.

Tabela 5.27: Tempos de execução com a aplicação `pmemd.cuda`. AMBER compilado com gcc-4.4.3, de acordo com a configuração *default*.

Modelo	Tempo (s)	*Speedup* (×)
SPSP	87	2,910
SPDP	92	2,752
DPDP	197	1,285
Cluster, NP = 16, IB (gcc-4.4.5)	253,156	---

A redução face ao melhor resultado no cluster é muito significativa, especialmente se atendermos ao facto de a comparação ser frente a um resultado obtido com a utilização de um *interconnect* muito rápido (*InfiniBand*). No entanto, notámos a utilização total de um dos quatro cores Xeon disponíveis até ao momento em que a aplicação terminou.

Relativamente ao *tradeoff* entre desempenho e precisão, os resultados fazem crer que o modelo misto apresenta o melhor compromisso, visto apenas demorar um pouco mais (+5,75%) que o modelo de precisão simples. Por esse motivo, o *speedup* obtido não se altera significativamente. O mesmo não podemos dizer para o modelo de precisão dupla; o resultado é agradável mas longe de espectacular, não ignorando a utilização de 25% do poder de computação (central) da máquina. No entanto esta será a real medida de comparação com o cluster, admitindo que o código não CUDA é inteiramente *double-precision*.

5.15 Experimentando outras Aplicações

Nas secções precedentes apresentámos e discutimos os resultados de vários ensaios realizados com a *suite* AMBER. Nas subsecções seguintes iremos averiguar se é possível melhorar o desempenho de aplicações semelhantes, recorrendo às mesmas técnicas que empregámos anteriormente; esperamos assim conseguir reunir um conjunto de linhas de orientação que permitam aos utilizadores dessas aplicações obterem melhores resultados e prescindirem de todo um trabalho de experimentação que de outra forma teriam de realizar.

[40] *Compile and Run.*

[41] Tipicamente a dupla precisão acarreta um custo correspondente ao dobro daquele com precisão simples.

[42] Existe uma excepção conforme descrito na página 215 do manual do AMBER.

5.15.1 NAMD

O NAMD[43] é um software para simulação de dinâmica molecular, desenvolvido no Instituto Beckman[44] em Illinois (EUA), que oferece um conjunto de funcionalidades semelhantes às apresentadas pelo AMBER. No entanto, o NAMD encontra-se disponível gratuitamente para o utilizador não comercial, contrariamente ao que se sucede com a *suite* AMBER, que tem um preço meramente simbólico.

Utilizaremos a versão 2.8, de Maio de 2011, compilada por nós a partir do seu código fonte. Não obstante o NAMD recorrer a uma linguagem paralela[45] que dispensa a utilização de MPI, optamos por deliberadamente ignorar tal facto; somente ao compilar a sua versão MPI tornamos possível uma comparação realista com os resultados obtidos com o AMBER. Para o fazer, usaremos um exemplo que encontrámos no site do software, correspondente a um vírus (STMV[46]) excedendo um milhão de átomos.

5.15.1.1 *Interconnects*

Começamos por executar dois testes sobre os 4 nós do cluster (NP = 16) utilizando a configuração *default* e o gcc-4.4.5, como habitual. Para o efeito considerámos a utilização de *InfiniBand* (IB) e *IP over Ethernet* (IPoE). A tabela seguinte apresenta os tempos registados.

Tabela 5.28: Tempos NAMD no *cluster* (NP = 16). Configuração *default* com gcc-4.4.5.

Opção	Tempo (s)	*Speedup* (×)
IP over Ethernet	986,38	---
InfiniBand	936,25	1,054

Os resultados atestam uma melhoria na ordem dos 5% ao usar *InfiniBand* em vez do *Gigabit Ethernet*. O resultado reflecte um comportamento muito diverso do exibido pelo AMBER, e pode ficar a dever-se tanto à natureza da simulação, distinta da anteriormente usada, como à forma como a aplicação distribui os dados pelos nós de computação.

5.15.1.2 Usando a melhor configuração com GCC

Um ponto de assumido interesse consiste em averiguar se as *flags* que determinámos como sendo a melhor configuração com GCC no caso do AMBER, fruto do estudo individual que apresentámos algumas secções antes, também produzem um resultado semelhante quando aplicadas ao NAMD. Em caso afirmativo, podemos extrapolar que tais flags poderão, eventualmente, produzir resultados em aplicações semelhantes. A tabela 5.30 dá-nos a resposta.

[43] `http://www.ks.uiuc.edu/Research/namd/`

[44] `http://www.beckman.illinois.edu/`

[45] Charm++, disponível em `http://charm.cs.uiuc.edu/`.

[46] O *Satellite Tobacco Mosaic Virus* (STMV) é um vírus pequeno que amplia os efeitos do *Tobacco Mosaic Virus* (TMV), um outro vírus que afecta particularmente a planta do tabaco. Bastante informação pode ser encontrada em `http://www.ks.uiuc.edu/Research/STMV/` e `http://en.wikipedia.org/wiki/Tobacco_mosaic_virus`.

Tabela 5.29: Tempos NAMD no *cluster* (NP = 16). Melhor configuração com gcc-4.4.5.

Opção	Tempo (s)	*Speedup* (×)
IP over Ethernet	945,42	---
InfiniBand	885,96	1,067

Tendo em conta os valores apresentados, registamos melhorias de 4,15% e 5,68%, respectivamente para IPoE e IB, em relação aos resultados anteriores. Verificamos também um acréscimo ligeiro do *speedup*, favorável ao *InfiniBand*. Concluímos, portanto, que as flags que utilizámos no AMBER são capazes de proporcional melhorias comparáveis no NAMD[47].

5.15.1.3 Outros compiladores

Não poderíamos deixar de experimentar outros compiladores junto do NAMD. Neste caso, foi-nos possível compilar com sucesso recorrendo ao Open64, o que se revela uma mais-valia já que nos permite comparar o resultado com aquele obtido pelo Intel, também considerado, suportando algumas conclusões que apresentaremos oportunamente. Primeiramente exibimos os resultados.

Tabela 5.30: Tempos NAMD no *cluster* (NP = 16), *InfiniBand*. NAMD compilado de acordo com a configuração *default*.

Opção	Tempo (s)	*Speedup* (×)
gcc-4.1.2	965,17	---
gcc-4.4.5	936,25	1,031
Intel	959,79	1,006
Open64 (*)	866,91	1,113

(*) Note-se que não há suporte para Open64. Por essa razão também não há *default* para Open64. As flags que usámos foram: `-O3 -march=auto -ffast-math`.

O resultado obtido com a *suite* da AMD captura de imediato a nossa atenção, ao conseguir superar a compilação *default* com gcc-4.1.2 em 10,18%. Contrariamente, os compiladores da Intel continuam a mostrar-se incapazes de melhorar o resultado, o que de certa forma nos faz acreditar que os resultados obtidos com este compilador junto do AMBER estão correctos. A nossa convicção não é que *kit* da Intel seja objectivamente inferior a outro qualquer considerado, pelo contrário[48]; acreditamos que a questão essencial aqui é a plataforma com a qual trabalhamos[49]. Nesse sentido, não temos muitas dúvidas que numa plataforma Intel o resultado obtido pelos seus compiladores seria, provavelmente, muito diferente. Infelizmente não estamos em condições de testá-lo, tal como não estamos em relação ao Open64 noutro contexto. Todavia, ponderando todos os aspectos referidos, deixamos a sugestão de

[47] Na verdade, podemos afirmar que estes resultados são melhores do que fizemos crer. Acontece que a configuração *default* do NAMD inclui a utilização da flag `-ffast-math` que pode implicar a violação de standards IEEE respeitantes a operações matemáticas. Considerando que (a) essa flag pode trazer consigo uma melhoria de desempenho, em sacrifício da precisão e (b) que as flags que utilizámos não incluem a anterior, por uma questão de coerência; então eventualmente poderíamos ter registado tempos mais favoráveis.

[48] Aliás nem poderíamos ignorar inúmeros relatos da comunidade dando conta de melhores resultados obtidos com compiladores Intel.

[49] Relembrando o nosso cluster é formado por processadores AMD.

ser utilizado o compilador do fabricante, pois acreditamos que será esse a causar o melhor resultado.

5.15.1.4 Ligação estática

Como tivemos oportunidade de observar, a ligação estática do AMBER não conseguiu superar o resultado obtido pela ligação dinâmica; se o leitor se recorda o resultado foi ligeiramente inferior até. Por essa razão, pretendemos averiguar se obtemos o mesmo comportamento com o NAMD. A tabela seguinte elucida-nos nesse aspecto.

Tabela 5.31: Tempos NAMD no *cluster* (NP = 16), *InfiniBand*.

Opção	Tempo (s)	*Speedup* (×)
gcc (*default*)	936,25	---
gcc (*static*)	2709,16	0,346

Desta vez, os resultados são incontestáveis: a utilização de ligação estática é muitíssimo desfavorável.

5.15.1.5 *Hugepages*

Para concluir a breve incursão com o NAMD, realizámos alguma experimentação com *hugepages*. Conduzimos duas experiências diferentes: com GCC, e usando o método `malloc`, fizemos três ensaios disponibilizando diferentes quantidades de páginas; com Open64, utilizando as facilidades que oferece directamente para *hugepages*, disponibilizámos 125 páginas a cada processo[50].

Tabela 5.32: Tempos NAMD no *cluster* (NP = 16), *InfiniBand*. *Speedups* em relação ao ensaio sem *hugepages* correspondente.

Opção	# *Hugepages*	Tempo (s)	*Speedup* (×)
gcc-4.4.5	500	928,48	1,008
	1000	1030,89	0,908
	1500	1085,19	0,863
Open64	125	870,05	0,996

A tabela 5.32, exibindo os resultados com GCC, permite-nos concluir que a disponibilização e respectiva utilização[51] de um maior número de páginas leva a uma degradação da performance.

[50] O Open64 permite que o façamos. Tal é extremamente útil, pois impede que um único processo possa consumir todas páginas aos demais, o que possivelmente levaria a um mau resultado.

[51] Observámos que as páginas eram, de facto, utilizadas pela aplicação. Tendo em conta os valores apresentados, será fácil concluir que esta simulação com NAMD utiliza consideravelmente mais memória que aquela utilizada com o AMBER. Ainda assim, tal não tornou as *hugepages* interessantes para o NAMD.

Capítulo 6

Conclusões

Tendo apresentado todos os resultados referentes aos estudos que nos propusemos a realizar, é tempo de relembrar as metas que foram alcançadas. Neste capítulo final, proporcionamos ao leitor uma forma de acesso rápido a todas as conclusões que fomos reunindo ao longo do nosso percurso em busca de uma melhor performance; ao condensá-las num curto espaço, colocamos o leitor em condições de apreciar as contribuições do trabalho e as recomendações que deixamos.

Em primeiro lugar permitam-nos destacar alguns dos factos por nós apurados e a sua pertinência:

- Determinámos que a utilização de uma versão recente do(s) compilador(es), em substituição da que é fornecida com a distribuição do SO é de extrema importância - especialmente quando esta última é razoavelmente antiga, como acontece em distribuições baseadas no Red Hat (e.g., CentOS e Scientific Linux), muito usadas em ambientes HPC;

- Conseguimos identificar – usando uma heurística simples – um conjunto de opções de compilação que produzem resultados similares (bons) com um variado leque de aplicações;

- Com grande satisfação constatámos ser possível obter uma melhoria considerável ao instalar um protocolo de baixa latência sobre *Ethernet*. Tal significa que um utilizador pode (a) retirar melhor partido do hardware *Gigabit Ethernet* e consequentemente (b) ponderar melhor a aquisição de um *interconnect* mais rápido;

- Provámos ser possível um incremento adicional de performance através da simples afinação de um parâmetro do sistema (`rx-usecs`) em execuções suportadas em *Ethernet*;

- Mostrámos que ao utilizar uma máquina com uma configuração fortemente NUMA um utilizador pode ver a sua aplicação demorar muito mais do que demoraria se a topologia do sistema fosse considerada. Sugerimos soluções para contornar a questão, incluindo um procedimento que possibilita "reservar" processadores para uso exclusivo da aplicação em causa;

- Confirmámos ser possível obter um bom *speedup* ao usar uma arquitectura com GPUs;

- Observámos que utilizar as ferramentas de compilação disponibilizadas pelo fabricante do processador produz código mais rápido;

- Observámos que a ligação estática e a utilização de *hugepages* não deve ser algo a considerar em aplicações que não consumam muita memória pois o mais provável é serem prejudiciais.

A melhor forma de avaliar a contribuição de cada um dos pontos enunciados é respondendo honestamente à questão "Quais dos anteriores considera serem do conhecimento do típico utilizador de aplicações de dinâmica molecular?". Deixamos esse desafio ao leitor.

Prosseguimos, apresentando uma tabela que quantifica cada um dos pontos que listámos anteriormente. Assumimos que o caso base corresponde a um utilizador que instala o CentOS 5.5 e adiciona software via `yum`[1]. Nesta avaliação baseamo-nos nos resultados obtidos com o AMBER no *cluster*, sem compromisso das restantes conclusões que recordámos há instantes.

Tabela 6.1: Avaliação Final. Resultados em percentagem e em relação ao caso "Base" correspondente.

Caso	Tempo	*Speedup* (×)
Base (gcc-4.1.2) + GbE	3425,29	---
+ AMD ACML	3307,63	1,036
gcc-4.4.5	3337,09	1,026
+ conjunto de flags gcc	3220,31	1,064
+ open-mx	1294,84	2,645
+ rx-usecs = 10μs	1242,07	2,758
Base (gcc-4.1.2) + IB	558,69	---
gcc-4.4.5	507,51	1,101
+ conjunto de flags gcc	485,97	1,150

O *speedup* máximo alcançado foi de 2,758×, ou 7.048× se admitirmos que o utilizador realiza o *upgrade* para a tecnologia *InfiniBand*. É um valor bastante satisfatório, principalmente se considerarmos as assumidas dificuldades em optimizar uma aplicação sem alterar o seu código, e cuja configuração *default* já inclui o uso de optimizações agressivas[2]. Em relação ao NAMD, a outra aplicação que testamos, o único senão na construção de *guidelines* de optimização é ao nível do *interconnect*. Não havendo diferença significativa entre IPoE e IB, então deixa de fazer sentido considerar Open-MX e, por conseguinte, perde-se alguma margem de manobra.

Por último, algumas considerações pessoais sobre o trabalho realizado. Pelos factos que aqui enunciamos, estamos convictos que honrámos os objectivos que projectámos alcançar: apresentámos um estudo tão amplo e completo quanto possível, e obtivemos bons resultados ainda que em última análise o sucesso deste trabalho não se meça necessariamente pela quantificação da melhoria obtida, mas sim pela demonstração do que pode, ou não, ser feito. Não obstante, desejamos dizer que à melhoria objectiva introduzida poderíamos acrescentar o tempo que o utilizador poupará ao não ter de analisar todas as questões que abordámos ao longo desta agradável viagem; sentimos que tal pode ser facilmente ignorado[3].

[1] Com excepção do Open MPI.

[2] Fundamentalmente `-O3`. Este é outro factor que o leitor deve ter em conta ao apreciar as nossas contribuições.

[3] Cada resultado tem por detrás tempo de exploração de várias alternativas, muitas nem mencionadas por se terem revelado infrutíferas, configuração de software requerendo leitura dos seus manuais, preparação prévia para migração para máquinas reais dado a responsabilidade que acarreta, *scripting*, inúmeros *runs*, superação de erros, pesquisa, etc.

Capítulo 7

Trabalho Futuro

Como se referiu no Capítulo 1, este é um primeiro passo no sentido de definir uma metodologia que permita a um utilizador obter, de forma simples, ganhos de desempenho para as aplicações HPC que instala sem ter de ter conhecimentos avançados na área do Software.

Se do ponto de vista das arquitecturas, software e *middleware* se conseguiu experimentar todos aqueles considerados importantes, o mesmo não aconteceu quanto ao perfil computacional das simulações tentadas, nem sequer ao leque de aplicações utilizadas.

Sendo assim, os próximos passos, já definidos e prontos a explorar são:

- Aplicação das heurísticas a uma aplicação de cálculo de estruturas usando o método de elementos finitos;
- Exploração de um leque alargado de diferentes simulações de MD que incluam diferentes perfis computacionais no que se refere ao uso de memória e da infra-estrutura de comunicação (já que sabemos de antemão que em termos de CPU o uso é... máximo). Para cada caso será necessário criar os ficheiros de input nos 3 formatos, PMEMD, SANDER e NAMD, num total estimado de 3×3×3 variantes.

Bibliografia

[1] A brief summary of hugetlbpage support in the Linux kernel. http://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[2] Amber 11 Users' Manual. http://ambermd.org/doc11/Amber11.pdf.

[3] CUBLAS User Guide. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs%/CUBLAS_Library.pdf.

[4] CUDA Reference Manual. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs%/CUDA_Toolkit_Reference_Manual.pdf.

[5] GPU Computing SDK code samples. http://developer.download.nvidia.com/compute/cuda/3_2_prod/sdk/gpucompu%tingsdk_3.2.16_linux.run.

[6] Intel 64 and IA-32 Architectures Software Developer's Manuals. http://www.intel.com/products/processor/manuals/.

[7] LAPACK Users' Guide. http://www.netlib.org/lapack/lug/.

[8] libhugetlbfs. http://libhugetlbfs.sourceforge.net/.

[9] OpenMP Specification. http://www.openmp.org/mp-documents/spec30.pdf.

[10] Blaise Barney. Message Passing Interface (MPI). https://computing.llnl.gov/tutorials/mpi/. Lawrence Livermore National Laboratory.

[11] R. Buyya. High Performance Cluster Computing: Architectures and systems. High Performance Cluster Computing. Prentice Hall PTR, 1999.

[12] B. Chapman, G. Jost, and R. Pas. Using OpenMP: portable shared memory parallel programming. Number v. 10 in Scientific and engineering computation. MIT Press, 2007.

[13] Paulo Afonso Cruzeiro, Leonor e Lopes. Parallel computing and protein design. In Proceedings of the 4th Iberian Grid Infrastructure Conference, Braga, Portugal, 2010.

[14] H. El-Rewini and M. Abd-El-Barr. Advanced computer architecture and parallel processing. Wiley series on parallel and distributed computing. Wiley, 2005.

[15] M. J. Flynn. Some computer organizations and their effectiveness. IEEE Transactions on Computers, C-21(9):948–960, 1972.

[16] Free Software Foundation. GNU Binary Utilities — objdump documentation. http://sourceware.org/binutils/docs-2.21/binutils/objdump.html#objdump.

[17] Free Software Foundation, Inc. GCC 4.1.2 Manual. http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/.

[18] Free Software Foundation, Inc. GCC 4.4.5 Manual. http://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/.

[19] Stefan Goedecker and A. Hoisie. Performance optimization of numerically intensive codes. Software, environments, tools. Society for Industrial and Applied Mathematics, 2001.

[20] J.L. Hennessy, D.A. Patterson, and A.C. Arpaci-Dusseau. Computer architecture: a quantitative approach. Number v. 1 in The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2007.

[21] Intel. Optimization Notice. http://software.intel.com/en-us/articles/optimization-notice/.

[22] D. Kirk, W.W. Hwu, and W. Hwu. Programming massively parallel processors: a hands-on approach. Applications of GPU Computing Series. Morgan Kaufmann Publishers, 2010.

[23] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pages 19–25, December 1995.

[24] J. Sanders and E. Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming. Pearson Education, 2010.

[25] T. Shanley, J. Winkles, and Inc MindShare. InfiniBand network architecture. PC system architecture series. Addison-Wesley, 2003.

[26] A. Silberschatz, P.B. Galvin, and G. Gagne. Operating system concepts. Wiley, 2009.

[27] Robert S. Sinkovits and Jerry P Greenberg. Optimizing amber for the cray t3e. In CUG 1999 Spring Proceedings, California, 1999.

[28] M. *et al* Snir. MPI: The Complete Reference. Number v. 1. MIT Press, 1998.

[29] V. Wojcik. Lecture notes. http://www.cosc.brocku.ca/~vwojcik/3P93/notes/2-Taxonomy.pdf, 2003. Brock University, Ontario, Canada.

[30] D.C. Young. Computational drug design: a guide for computational and medicinal chemists. John Wiley & Sons, 2009.

Apêndice A

μ-*Benchmarks* desenvolvidos

A.1 Exemplo 1: matrix_mul.c

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_WIDTH 1000000


void
MatrixMultiplication (float *M, float *N, float *P, int width)
{
  int i, j, k;

  for (i = 0; i < width; i++)
    {
      for (j = 0; j < width; j++)
{
  float sum = 0;

  for (k = 0; k < width; k++)
    sum += M[i * width + k] * N[k * width + j];

  P[i * width + j] = sum;
}
    }
}

void
usage (char *pname)
{
  printf ("Usage: %s WIDTH\nMultiplies two square matrices of order WIDTH.",
  pname);
  printf ("These matrices are randomly initialized at runtime.\n\n");
  printf ("\tWIDTH\tan integer between 0 and %d\n", MAX_WIDTH);
}



int
main (int argc, char **argv)
{
  if (argc != 2)
    {
```

```
      usage (argv[0]);
      return EXIT_SUCCESS;
    }

  int width = atoi (argv[1]);

  if (width < 0 || width > MAX_WIDTH)
    {
      usage (argv[0]);
      return EXIT_SUCCESS;
    }

  // Allocate three square matrices of order 'width'
  size_t size = width * width * sizeof (float);
  float *M = (float *) malloc (size);
  float *N = (float *) malloc (size);
  float *P = (float *) malloc (size);

  if (M == NULL || N == NULL || P == NULL)
    {
      printf ("Error allocating memory. Aborting...\n");
      return EXIT_FAILURE;
    }

  // Initialization
  int i;
  for (i = 0; i < width * width; i++)
    {
      M[i] = rand () / (float) RAND_MAX;
      N[i] = rand () / (float) RAND_MAX;
    }

  MatrixMultiplication (M, N, P, width);

  free (M);
  free (N);
  free (P);

  return EXIT_SUCCESS;
}
```

A.2 Exemplo 2: matrix_mul_cublas.c

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>

#define MAX_WIDTH 1000000

/* Adaptado de:
http://developer.download.nvidia.com/compute/cuda/sdk/Projects/simpleCUBLAS.tar.gz
*/

void
usage (char *pname)
{
  printf ("Usage: %s WIDTH\nMultiplies two square matrices of order WIDTH.",
```

```
  pname);
  printf ("These matrices are randomly initialized at runtime.\n\n");
  printf ("\tWIDTH\tan integer between 0 and %d\n", MAX_WIDTH);
}



int
main (int argc, char **argv)
{
  if (argc != 2)
    {
      usage (argv[0]);
      exit (EXIT_SUCCESS);
    }

  int width = atoi (argv[1]);

  if (width < 0 || width > MAX_WIDTH)
    {
      usage (argv[0]);
      exit (EXIT_SUCCESS);
    }

  int i;
  float *h_M;
  float *h_N;
  float *h_P;
  float *d_M = 0;
  float *d_N = 0;
  float *d_P = 0;
  float alpha = 1.0f;
  float beta = 0.0f;
  int n2 = width * width;
  cublasStatus status;

  /* Initialize CUBLAS */
  status = cublasInit ();
  if (status != CUBLAS_STATUS_SUCCESS)
    {
      printf ("Error initializing CUBLAS. Aborting ...\n");
      return EXIT_FAILURE;
    }

  /* Allocate host memory for the matrices */
  h_M = (float *) malloc (n2 * sizeof (h_M[0]));
  if (h_M == 0)
    {
      fprintf (stderr, "!!!! host memory allocation error (A)\n");
      return EXIT_FAILURE;
    }
  h_N = (float *) malloc (n2 * sizeof (h_N[0]));
  if (h_N == 0)
    {
      fprintf (stderr, "!!!! host memory allocation error (B)\n");
      return EXIT_FAILURE;
    }
  h_P = (float *) malloc (n2 * sizeof (h_P[0]));
```

```
if (h_P == 0)
  {
    fprintf (stderr, "!!!! host memory allocation error (C)\n");
    return EXIT_FAILURE;
  }

/* Fill the matrices with test data */
for (i = 0; i < n2; i++)
  {
    h_M[i] = rand () / (float) RAND_MAX;
    h_N[i] = rand () / (float) RAND_MAX;
    h_P[i] = 0.0f;
  }

/* Allocate device memory for the matrices */
status = cublasAlloc (n2, sizeof (d_M[0]), (void **) &d_M);
if (status != CUBLAS_STATUS_SUCCESS)
  {
    fprintf (stderr, "!!!! device memory allocation error (A)\n");
    return EXIT_FAILURE;
  }
status = cublasAlloc (n2, sizeof (d_N[0]), (void **) &d_N);
if (status != CUBLAS_STATUS_SUCCESS)
  {
    fprintf (stderr, "!!!! device memory allocation error (B)\n");
    return EXIT_FAILURE;
  }
status = cublasAlloc (n2, sizeof (d_P[0]), (void **) &d_P);
if (status != CUBLAS_STATUS_SUCCESS)
  {
    fprintf (stderr, "!!!! device memory allocation error (C)\n");
    return EXIT_FAILURE;
  }

/* Initialize the device matrices with the host matrices */
status = cublasSetVector (n2, sizeof (h_M[0]), h_M, 1, d_M, 1);
if (status != CUBLAS_STATUS_SUCCESS)
  {
    fprintf (stderr, "!!!! device access error (write A)\n");
    return EXIT_FAILURE;
  }
status = cublasSetVector (n2, sizeof (h_N[0]), h_N, 1, d_N, 1);
if (status != CUBLAS_STATUS_SUCCESS)
  {
    fprintf (stderr, "!!!! device access error (write B)\n");
    return EXIT_FAILURE;
  }
status = cublasSetVector (n2, sizeof (h_P[0]), h_P, 1, d_P, 1);
if (status != CUBLAS_STATUS_SUCCESS)
  {
    fprintf (stderr, "!!!! device access error (write C)\n");
    return EXIT_FAILURE;
  }

/* Clear last error */
cublasGetError ();

/* Performs operation using cublas */
```

```
  cublasSgemm ('n', 'n', width, width, width, alpha, d_M, width, d_N, width,
       beta, d_P, width);
  status = cublasGetError ();
  if (status != CUBLAS_STATUS_SUCCESS)
    {
      fprintf (stderr, "!!!! kernel execution error.\n");
      return EXIT_FAILURE;
    }

  /* Allocate host memory for reading back the result from device memory */
  h_P = (float *) malloc (n2 * sizeof (h_P[0]));
  if (h_P == 0)
    {
      fprintf (stderr, "!!!! host memory allocation error (C)\n");
      return EXIT_FAILURE;
    }

  /* Read the result back */
  status = cublasGetVector (n2, sizeof (h_P[0]), d_P, 1, h_P, 1);
  if (status != CUBLAS_STATUS_SUCCESS)
    {
      fprintf (stderr, "!!!! device access error (read C)\n");
      return EXIT_FAILURE;
    }

  /* Memory clean up */
  free (h_M);
  free (h_N);
  free (h_P);
  status = cublasFree (d_M);
  if (status != CUBLAS_STATUS_SUCCESS)
    {
      fprintf (stderr, "!!!! memory free error (A)\n");
      return EXIT_FAILURE;
    }
  status = cublasFree (d_N);
  if (status != CUBLAS_STATUS_SUCCESS)
    {
      fprintf (stderr, "!!!! memory free error (B)\n");
      return EXIT_FAILURE;
    }
  status = cublasFree (d_P);
  if (status != CUBLAS_STATUS_SUCCESS)
    {
      fprintf (stderr, "!!!! memory free error (C)\n");
      return EXIT_FAILURE;
    }

  /* Shutdown */
  status = cublasShutdown ();
  if (status != CUBLAS_STATUS_SUCCESS)
    {
      fprintf (stderr, "!!!! shutdown error (A)\n");
      return EXIT_FAILURE;
    }

  return EXIT_SUCCESS;
}
```

A.3 Exemplo 3: matrix_mul_cuda.c

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

#define MAX_WIDTH 1000000


__global__ void
MatrixMultiplication (float *M, float *N, float *P, int width)
{
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float sum = 0;

  for (int k = 0; k < width; k++)
    {
      sum += M[ty * width + k] * N[k * width + tx];
    }

  P[ty * width + tx] = sum;
}

void
usage (char *pname)
{
  printf ("Usage: %s WIDTH\nMultiplies two square matrices of order WIDTH.",
  pname);
  printf ("These matrices are rad_Nomly initialized at runtime.\n\n");
  printf ("\tWIDTH\tan integer between 0 ad_N %d\n", MAX_WIDTH);
}



int
main (int argc, char **argv)
{
  if (argc != 2)
    {
      usage (argv[0]);
      return EXIT_SUCCESS;
    }

  int width = atoi (argv[1]);

  if (width < 0 || width > MAX_WIDTH)
    {
      usage (argv[0]);
      return EXIT_SUCCESS;
    }

  float *h_M, *d_M, *h_N, *d_N, *h_P, *d_P;

  // Allocate three square matrices of order 'width'
  size_t size = width * width * sizeof (float);
  h_M = (float *) malloc (size);
```

```
  h_N = (float *) malloc (size);
  h_P = (float *) malloc (size);

  if (h_M == NULL || h_N == NULL || h_P == NULL)
    {
      printf ("Error allocating memory. Aborting...\n");
      return EXIT_FAILURE;
    }

  cudaMalloc ((void **) &d_M, size);
  cudaMalloc ((void **) &d_N, size);
  cudaMalloc ((void **) &d_P, size);

  // Initialization
  int i;
  for (i = 0; i < width * width; i++)
    {
      h_M[i] = rand () / (float) RAND_MAX;
      h_N[i] = rand () / (float) RAND_MAX;
    }

  cudaMemcpy ((void *) d_M, (const void *) h_M, size, cudaMemcpyHostToDevice);
  cudaMemcpy ((void *) d_N, (const void *) h_N, size, cudaMemcpyHostToDevice);

  dim3 dimBlock (width, width);
  dim3 dimGrid (1, 1);

  MatrixMultiplication <<< dimGrid, dimBlock >>> (d_M, d_N, d_P, width);

  cudaMemcpy ((void *) h_P, (const void *) d_P, size, cudaMemcpyDeviceToHost);

  free (h_M);
  free (h_N);
  free (h_P);

  cudaFree (d_M);
  cudaFree (d_N);
  cudaFree (d_P);

  return EXIT_SUCCESS;
}
```

A.4 Exemplo 4: matrix_mul_mpi.c

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define TAG1 1000
#define TAG2 1001

#define MASTER 0
#define MAX_WIDTH 1000000

#define NROWS(N,W,R) (W / N + (W % N > R))
```

```
void
MatrixMultiplication (float *M, float *N, float *P, int nrows, int width)
{
  int i, j, k;

  for (i = 0; i < nrows; i++)
    {
      for (j = 0; j < width; j++)
{
  float sum = 0;

  for (k = 0; k < width; k++)
    sum += M[i * width + k] * N[k * width + j];

  P[i * width + j] = sum;
}
    }
}

void
usage (char *pname)
{
  printf ("Usage: %s WIDTH\nMultiplies two square matrices of order WIDTH.",
  pname);
  printf ("These matrices are randomly initialized at runtime.\n\n");
  printf ("\tWIDTH\tan integer between 0 and %d\n", MAX_WIDTH);
}

int
main (int argc, char **argv)
{
  int numtasks, rank, rc;

  rc = MPI_Init (&argc, &argv);
  if (rc != MPI_SUCCESS)
    {
      printf ("Error starting MPI program. Terminating.\n");
      MPI_Abort (MPI_COMM_WORLD, rc);
    }

  MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  int width = 0;

  if (rank == MASTER)
    {
      if (argc != 2)
usage (argv[0]);
      else
{
  width = atoi (argv[1]);

  if (width < 0 || width > MAX_WIDTH)
    usage (argv[0]);
}
    }
```

```
  MPI_Bcast (&width, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if (width == 0)
    {
      MPI_Finalize ();
      return EXIT_SUCCESS;
    }

  int i;
  int nelems = width * width;
  int nrows = NROWS (numtasks, width, rank);
  size_t size = nelems * sizeof (float);
  size_t slice = nrows * width * sizeof (float);

  float *M = (float *) malloc (rank == 0 ? size : slice);
  float *N = (float *) malloc (size);
  float *P = (float *) malloc (rank == 0 ? size : slice);

  if (M == NULL || N == NULL || P == NULL)
    {
      printf ("Task %d: error allocating memory. Aborting ...\n", rank);

      MPI_Finalize ();
      return EXIT_FAILURE;
    }


  if (rank == MASTER)
    {
      for (i = 0; i < nelems; i++)
{
  M[i] = rand () / (float) RAND_MAX;
  N[i] = rand () / (float) RAND_MAX;
}
    }

  if (rank == MASTER)
    {
      int offset = width * nrows;

      for (i = 1; i < numtasks; i++)
{
  int nmemb = width * NROWS (numtasks, width, i);

  MPI_Send (M + offset, nmemb, MPI_FLOAT, i, TAG1, MPI_COMM_WORLD);

  offset += nmemb;
}
    }
  else
    {
      MPI_Recv (M, width * nrows, MPI_FLOAT, MASTER, TAG1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }

  MPI_Bcast (N, nelems, MPI_FLOAT, MASTER, MPI_COMM_WORLD);

  MatrixMultiplication (M, N, P, nrows, width);
```

```
  if (rank == MASTER)
    {
      int offset = width * nrows;

      for (i = 1; i < numtasks; i++)
{
  int nmemb = width * NROWS (numtasks, width, i);

  MPI_Recv (P + offset, nmemb, MPI_FLOAT, i, TAG2,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  offset += nmemb;
}
    }
  else
    {
      MPI_Send (P, width * nrows, MPI_FLOAT, MASTER, TAG2, MPI_COMM_WORLD);
    }

  MPI_Finalize ();

  free (M);
  free (N);
  free (P);

  return EXIT_SUCCESS;
}
```

A.5 Exemplo 5: matrix_mul_omp.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>


#define MAX_WIDTH 1000000
#define NUM_THREADS 4


void
MatrixMultiplication (float *M, float *N, float *P, int width)
{
  int i, j, k;

#pragma omp parallel for private(j,k) \
num_threads(NUM_THREADS) \
schedule(static)
  for (i = 0; i < width; i++)
    {
      for (j = 0; j < width; j++)
{
  float sum = 0;

  for (k = 0; k < width; k++)
    sum += M[i * width + k] * N[k * width + j];
```

```
  P[i * width + j] = sum;
}
    }
}

void
usage (char *pname)
{
  printf ("Usage: %s WIDTH\nMultiplies two square matrices of order WIDTH.",
  pname);
  printf ("These matrices are randomly initialized at runtime.\n\n");
  printf ("\tWIDTH\tan integer between 0 and %d\n", MAX_WIDTH);
}



int
main (int argc, char **argv)
{
  if (argc != 2)
    {
      usage (argv[0]);
      return EXIT_SUCCESS;
    }

  int width = atoi (argv[1]);

  if (width < 0 || width > MAX_WIDTH)
    {
      usage (argv[0]);
      return EXIT_SUCCESS;
    }

  // Allocate three square matrices of order 'width'
  size_t size = width * width * sizeof (float);
  float *M = (float *) malloc (size);
  float *N = (float *) malloc (size);
  float *P = (float *) malloc (size);

  if (M == NULL || N == NULL || P == NULL)
    {
      printf ("Error allocating memory. Aborting...\n");
      return EXIT_FAILURE;
    }

  // Initialization
  int i;
  for (i = 0; i < width * width; i++)
    {
      M[i] = rand () / (float) RAND_MAX;
      N[i] = rand () / (float) RAND_MAX;
    }

  MatrixMultiplication (M, N, P, width);

  free (M);
  free (N);
  free (P);
```

```
  return EXIT_SUCCESS;
}
```

A.6 util.c

```
#include <time.h>


struct timespec
diff (struct timespec ts1, struct timespec ts2)
{
  struct timespec tmp;

  if ((ts2.tv_nsec - ts1.tv_nsec) < 0)
    {
      tmp.tv_sec = ts2.tv_sec - ts1.tv_sec - 1;
      tmp.tv_nsec = 1000000000 + ts2.tv_nsec - ts1.tv_nsec;
    }
  else
    {
      tmp.tv_sec = ts2.tv_sec - ts1.tv_sec;
      tmp.tv_nsec = ts2.tv_nsec - ts1.tv_nsec;
    }

  return tmp;
}
```

A.7 Exemplo 6: vectorAdd.c

```
#include <stdio.h>
#include <time.h>


struct timespec diff (struct timespec ts1, struct timespec ts2);


void
f (float *m, float *n, float *p)
{
  p[0] = m[0] + n[0];
  p[1] = m[1] + n[1];
  p[2] = m[2] + n[2];
  p[3] = m[3] + n[3];
}

int
main ()
{
  struct timespec time, time1, time2;

  float m[4] = { 1, 2, 3, 4 };
  float n[4] = { 1, 2, 3, 4 };
  float p[4];

  clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &time1);
  f (m, n, p);
```

```
  clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &time2);

  time = diff (time1, time2);

  printf ("Elapsed time: %ld seconds %ld nanoseconds\n", time.tv_sec,
  time.tv_nsec);

  printf ("%f, %f, %f, %f\n", p[0], p[1], p[2], p[3]);

  return 0;
}
```

A.8 Exemplo 7: vectorAdd_sse.c

```
#include <stdio.h>
#include <time.h>
#include <xmmintrin.h>


struct timespec diff (struct timespec ts1, struct timespec ts2);


void
f (__m128 * m, __m128 * n, __m128 * p)
{
  *p = _mm_add_ps (*m, *n);
}

int
main ()
{
  __m128 m, n, p;
  struct timespec time, time1, time2;

  m = _mm_set_ps (4, 3, 2, 1);
  n = _mm_set_ps (4, 3, 2, 1);

  clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &time1);
  f (&m, &n, &p);
  clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &time2);

  time = diff (time1, time2);

  printf ("Elapsed time: %ld seconds %ld nanoseconds\n", time.tv_sec,
  time.tv_nsec);

  float *ptr = (float *) &p;

  printf ("%f, %f, %f, %f\n", ptr[0], ptr[1], ptr[2], ptr[3]);

  return 0;
}
```

A.9 Exemplo 8: linking.c

```
#include <math.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE 1000000


int
main ()
{
  int x;
  int *p, *q;

  x = atoi ("123");

  printf ("Olá!\n");

  x = rand ();

  p = (int *) malloc (ARRAY_SIZE * sizeof (int));

  puts ("Hello!\n");

  abs (-1);

  log (1024);

  q = (int *) calloc (ARRAY_SIZE, sizeof (int));

  for (x = 0; x < ARRAY_SIZE; x++)
    strcmp ("Isto é apenas um teste?", "Isto é apenas um teste!");

  memcpy ((void *) p, (void *) q, ARRAY_SIZE * sizeof (int));

  free (p);
  free (q);

  exit (EXIT_SUCCESS);
}
```

A.10 Makefile

```
CFLAGS = -Wall -O2
CUDA = -I /usr/local/cuda/include/ -L /usr/local/cuda/lib/ -l cublas


all: cublas cuda linking mpi omp seq sse

cublas: matrix_mul_cublas.c
nvcc $(CUDA) -o matrix_mul_cublas matrix_mul_cublas.c

cuda: matrix_mul_cuda.cu
nvcc -o matrix_mul_cuda matrix_mul_cuda.cu

linking: linking.c
cc $(CFLAGS) -static -o static linking.c
cc $(CFLAGS) -o dynamic linking.c
```

```
mpi: matrix_mul_mpi.c
mpicc $(CFLAGS) -o matrix_mul_mpi matrix_mul_mpi.c

omp: matrix_mul_omp.c
cc $(CFLAGS) -fopenmp -o matrix_mul_omp matrix_mul_omp.c

seq: matrix_mul.c
cc $(CFLAGS) -o matrix_mul matrix_mul.c

sse: vectorAdd.c vectorAdd_sse.c util.c
cc $(CFLAGS) -lrt -o vectorAdd util.c vectorAdd.c
cc $(CFLAGS) -lrt -mfpmath=sse -msse -o vectorAdd_sse util.c \
vectorAdd_sse.c

clean:
rm -f matrix_mul_cublas matrix_mul_cuda static dynamic \
matrix_mul_mpi matrix_mul_omp matrix_mul vectorAdd vectorAdd_sse
```

Apêndice B

Resultados dos μ-*Benchmarks*

Por forma a comparar o desempenho das diferentes implementações consideradas[1], entre si e quando comparadas com a versão sequencial, foram registados os tempos de execução obtidos em três ensaios.

Antes de apresentar os resultados, é fundamental introduzir as características da máquina onde foram compilados e executados os programas em causa.

Configuração:

Intel Core2 Duo CPU T9400 @ 2,53GHz (20,24 GFLOPS)
3GB RAM (6,4 GB/s)
GeForce 9600M GT 512 MB (12,8 GB/s, 32 CUDA Cores @ 500 MHz, 120 GFLOPS)

Ambiente (x86):

Linux 2.6.34.7
gcc 4.4.5
MPICH2 1.2.1p1
Nvidia Driver 260.19.14

A tabela B.1 apresenta o tempo médio de execução e *speedup* alcançado (relativo à versão sequencial) para as quatro implementações da aplicação de multiplicação de duas matrizes. Os valores obtidos dizem respeito a execuções com matrizes quadradas de ordem 1000, inicializadas de forma aleatória.

Tabela B.1: Tempos de execução, em segundos, para as versões Sequencial, CUBLAS, CUDA, MPI e OMP.

	Sequencial	CUBLAS	CUDA	MPI	OMP
Tempo (s)	4,841	0,495	0,163	2,903	2,749
Speedup (×)	---	9,78	29,7	1,67	1,76

Repare-se que ambas as implementações executadas exclusivamente pelo CPU apresentam *speedups* bastante próximos do máximo teórico de 2×, considerando a configuração acima indicada. Observa-se ainda que a versão OMP supera marginalmente a versão MPI, o que se poderá atribuir ao *overhead* inerente à troca de mensagens realizada na última, por oposição

[1] Ver secção A, Apêndices.

à utilização de memória partilhada que se verifica na primeira[2].

É de realçar o resultado expressivo obtido pelas versões CUDA, ainda que julgue sensato admitir que se seria conveniente definir uma medida rigorosa de comparação entre o GPU e o CPU em questão. Caso contrário, torna-se difícil avaliar a qualidade da paralelização, para além do que é óbvio. Uma medida minimalista de comparação poderá ser aquela dada pelo produto dos rácios entre as larguras de banda das memórias (em GB/s) e entre as capacidades de cálculo (em GFLOPS), o que neste caso nos faria esperar um *speedup* de 10× para as aplicações executadas, na sua quase totalidade, pelo GPU. No entanto, e na prática, parece ser suficiente saber qual a melhoria de performance inerente à utilização do GPU em relação ao CPU disponível.

Estes resultados são extremamente significativos no contexto deste trabalho, já que fundamentam o nosso apetite pela computação paralela. Além disso, e não obstante os programas considerados serem muito simples, percebemos que a paralelização se traduz na utilização de recursos que, de outra forma, estariam a ser desperdiçados.

[2]E também pelo facto de termos apenas criado dois *threads*.